Foundations and Trends® in
Databases
Vol. 4, No. 4 (2011) 295–405

DOI: 10.1561/1900000020

now
the essence of knowledge

Materialized Views

By Rada Chirkova and Jun Yang

Contents

Foundations and Trends® in
Databases
Vol. 4, No. 4 (2011) 295–405

DOI: 10.1561/1900000020

now
the essence of knowledge

Materialized Views

Rada Chirkova[1] and Jun Yang[2]

[1] *North Carolina State University, Department of Computer Science, Raleigh, North Carolina, 27695-8206, USA, chirkova@csc.ncsu.edu*
[2] *Duke University, Department of Computer Science, Durham, North Carolina, 27708-0129, USA, junyang@cs.duke.edu*

Abstract

Materialized views are queries whose results are stored and maintained in order to facilitate access to data in their underlying base tables. In the SQL setting, they are now considered a mature technology implemented by most commercial database systems and tightly integrated into query optimization. They have also helped lay the foundation for important topics such as information integration and data warehousing. This monograph provides an introduction and reference to materialized views. We cover three fundamental problems: (1) maintaining materialized views efficiently when the base tables change, (2) using materialized views effectively to improve performance and availability, and (3) selecting which views to materialize. We also point out their connections to a few other areas in database research, illustrate the benefit of cross-pollination of ideas with these areas, and identify several directions for research on materialized views.

1

Introduction

Materialized views are a natural embodiment of the ideas of *precomputation* and *caching* in databases. Instead of computing a query from scratch from base data, a database system can use results that have already been computed, stored, and maintained. The ability of materialized views to speed up queries benefit most database applications, ranging from traditional querying and reporting to web database caching [255], online analytical processing [89], and data mining [208, 367]. By reducing dependency on the availability of base data, materialized views have also laid much of the foundation for information integration [45, 139, 270] and data warehousing [89, 242] systems. Because of their wide applicability, materialized views are a well-studied topic with both a rich research literature and mature commercial implementations. Our goal of this monograph is to provide an accessible introduction and reference to this topic, explain its core ideas, highlight its recent developments, and point out its sometimes subtle connections to other research topics in databases.

Background A database *view* is defined by a query. When evaluated, this *view definition query* returns the contents of the view. Database

users can pose queries or define other views over views just as they can over regular database tables. Conceptually, when the database system executes a query involving views, references to views are replaced by their definition queries (recursively, if a view is defined using other views), yielding a final query involving only regular "base tables."

Example 1.1 (Adapted from [188]). Consider a retailer with multiple stores across the globe. The stores are grouped into multiple geographic regions for administrative and accounting purposes. The retailer consolidates its inventory and sales information across all stores into a single relational database for auditing and analysis purposes. Consider some of the tables in such a database and their cardinality:

- `pos(itemID, storeID, date, qty, price)`, with one billion (10^9) rows, records point-of-sale transactions. There is one row for every item sold in a transaction, with the ID of item, the ID of the store selling it, the date of sale, the quantity sold, and the unit price.
- `stores(storeID, city, region)`, with 100 rows, records information about each store: namely, its ID, city, and region.
- `items(itemID, name, category, cost)`, with 50,000 rows, records information about each item: namely, its ID, name, product category, and cost per unit.

The following view, defined over `pos`, computes the total sales revenue generated for each item by each store:

```
CREATE VIEW TotalByItemStore(itemID,storeID,total) AS
  SELECT itemID, storeID, SUM(qty*price)
  FROM pos GROUP BY itemID, storeID;
```

Suppose a business analyst wants to know the total revenue generated by each store for each item category. This query can be written against the above view as:

```
SELECT storeID, category, SUM(total)              -- (Q1v)
FROM TotalByItemStore, items
```

```
WHERE TotalByItemStore.itemID = items.itemID
GROUP BY storeID, category;
```

When evaluating this query, the database system conceptually expands it to the following equivalent query involving only base tables:

```
SELECT storeID, category, SUM(qty*price)            -- (Q1)
FROM pos, items
WHERE pos.itemID = items.itemID
GROUP BY storeID, category;
```

Traditionally, views are "virtual" — the database system stores their definition queries but not their contents. Virtual views are often used to control access and provide alternative interfaces to base tables. They also support *logical data independence*: when the base table schema changes, views can be redefined to use the new schema, so application queries written against these views will continue to function.

Over the years, however, the concept and practice of *materialized views* have steadily gained importance. We materialize a view by storing its contents (though many cases call for alternative materialization strategies; see Section 2.3). Once materialized, a view can facilitate queries that use it (or can be rewritten to use it), when the base tables are expensive or even unavailable for access.

Example 1.2. Continuing with Example 1.1, suppose we materialize the view `TotalByItemStore`. Now, query `(Q1v)` can be evaluated by joining `items` with the materialized contents of `TotalByItemStore`. This evaluation strategy is more efficient than joining `items` and `pos`, because `TotalByItemStore` has up to $50{,}000 \times 100 = 5 \times 10^6$ rows, compared with 10^9 rows in `pos`.

Although `(Q1)` is not originally written over `TotalByItemStore`, it is possible to recognize that `(Q1)` can be rewritten as `(Q1v)` to take advantage of the materialized `TotalByItemStore`.

Key Problems Example 1.2 above illustrates one important question in the study of materialized views: how to *answer queries using views*, especially when the queries are not originally written in terms of the materialized views. The next natural question is, given a database workload (queries and modifications) as well as resource and performance requirements, how to *select what views to materialize* in the first place. Instead of relying on database administrators and application developers to answer these questions in an ad hoc fashion, we prefer a more systematic and automatic approach.

Materialized views are not free. Not only do they take additional space, but they also require maintenance: as base tables change, the materialized view contents become outdated. Thus, a third important question is how to *maintain materialized views* to keep them up to date with respect to the base tables. The most straightforward way to maintain a materialized view is to recompute its definition query over the new database state whenever the base tables change. However, in practice, since the numbers of rows changed are often small compared with the sizes of the entire base tables, *incremental view maintenance* — the practice of computing and applying only incremental changes to materialized views induced by base table changes — may work better than recomputation.

Example 1.3. Continuing with Examples 1.1 and 1.2, suppose that five rows have been inserted into base table `pos` as the result of a sale transaction δ involving five different items at a particular store. Recomputing the materialized view `TotalByItemStore` from scratch would be both unnecessary and expensive, because most of its 5×10^6 rows are not affected by δ. With incremental maintenance, loosely speaking, we only need to identify the five affected rows in `TotalByItemStore` and increment their `total` by `qty*price` of their corresponding new `pos` rows inserted by δ.

To recap, the examples above reveal three key problems concerning materialized views: how to maintain them (*view maintenance*), how to use them (*answering queries using views*), and how to select them (*view*

selection). Solutions and techniques developed for these questions over the years have made materialized views an indispensable technology that greatly enhances the performance and features of database systems and many data-intensive applications, such as those mentioned in the opening of this section. Most commercial database systems now offer built-in support of materialized views; for other systems there exist popular recipes for "simulating" support of materialized views.

The ideas underlying materialized views are simple: e.g., precomputation, caching, and incremental evaluation. However, the great database tradition of *declarative querying* is what distinguishes materialized views from generic applications of these ideas in other contexts, and makes materialized views especially interesting, powerful, and challenging at the same time. Thanks to standardized, declarative database languages with clean semantics, study of materialized views has generated a rich body of theory and practice, aimed at providing efficient, effective, automated, and general solutions to the three key problems above.

Scope and Organization There is a vast body of literature on materialized views dating back to 1980s, not to mention work related to or influenced by it. There have also been other authoritative references to the topic, most notably the 1999 book edited by Gupta and Mumick [188], the 2001 survey by Halevy [203] on answering queries using views, as well as relevant entries in the recently compiled *Encyclopedia of Database Systems* [291]. We intend this monograph to serve as an accessible introduction and reference to the topic of materialized views for database researchers. In addition to covering the core ideas behind materialized views, we will highlight recent developments (especially those since 2000), and discuss connections to other more recent research topics in databases. This monograph is a more of a pedagogical text than a manual: given a problem, instead of presenting one definitive solution (which in many cases may not be clear or even exist), we walk the readers through the line of reasoning and research developments leading to better understanding of the problem. Therefore, this monograph should be used as a companion to, rather than a substitute for, the literature on materialized views.

The breadth of work on materialized views is as daunting as its depth. Different data models and query languages — object-oriented, semistructured, spatiotemporal, streaming, probabilistic, just to name a few — give rise to a multitude of problem settings that sometimes call for specialized techniques. To make this monograph approachable and focused, we limit its scope mostly to nonrecursive SQL views; we also assume that readers are familiar with relational and bag algebras (see standard database textbooks such as [159, 335], or, for quick reference, [370] and [174], respectively). Our hope is that the core ideas we cover will help readers in further exploring other problem settings.

As mentioned earlier, materialized views now have mature implementations in most commercial database systems. In fact, the database industry has contributed significantly — in many cases as leaders — to the research literature. Written primarily with a research audience in mind, this monograph focuses on the research literature (including contributions from the industry) rather than the product specifics. While we give a high-level overview of commercial implementations, we offer no in-depth comparison of product features.

We note that materialized views are but one form of *derived data* — the result of applying some transformation, structural or computational, to base data. The use of derived data to facilitate access to base data is a recurring theme in computer science. Besides materialized views, other examples include caches, replicas, indexes, and synopses [16, 123]. Despite differences in the form, complexity, and precision of derived data, the three fundamental questions remain: how to use derived data, what to maintain as derived data, and how to maintain them. Oftentimes, ideas and techniques developed for one form of derived data can be adapted and applied to another setting with interesting benefits. The repertoire of techniques for materialized views has been enriched by ideas from other forms of derived data. At the same time, many research areas, old and new alike, have drawn insights from materialized views, implicitly or explicitly. This monograph will highlight a few examples of such cross-pollination.

In the remainder of this monograph, Section 2 covers the *view maintenance* problem. Section 3 covers the *view use* problem. Section 4

covers the *view selection* problem. Section 5 explores connections between materialized views and a few other topics. Section 6 concludes with our perspectives on the current state and future directions of the study of materialized views.

2

Maintaining Materialized Views

We represent a materialized view by a pair (Q_v, T_v), where Q_v denotes the view definition query and T_v denotes the currently materialized view contents. Suppose T_v is currently up to date; i.e., $T_v = Q_v(D)$, where D denotes the current state of the database. As discussed in Section 1, we need to maintain the materialized view when base tables change. More specifically, when the underlying database undergoes a modification — in general consisting of a sequence of insertions, deletions, and updates to base tables — that changes the database state from D to D', we need to update T_v to $Q_v(D')$. This setting, illustrated in Example 1.3, is what we typically mean by "view maintenance."

Broadly speaking, however, several other situations also require "maintenance":

- When the view definition changes from Q_v to Q'_v, we need to update T_v to $Q'_v(D)$. This problem is called *view adaptation*; see survey by Ross [338], and more recent work by Green and Ives [175]. The key is to find a way to leverage the previously materialized result $T_v = Q_v(D)$ to compute $Q'_v(D)$, which is related to the problem of answering queries using views (Section 3.1).

- When the underlying database undergoes a schema change, we may need to change the view definition Q_v so it remains valid; we then need to update T_v accordingly. This problem has been considered by Rundensteiner et al. in [311] and a series of follow-up work (see also Section 2.5.1).
- If we carry the idea of logical data independence to its full extent, a view should behave like a regular table, so its contents can be modified. When a user modifies the view contents from T_v to T'_v, we could determine a new database state D' such that $T'_v = Q_v(D')$. In general, however, there may be many or none such database states. Traditionally, database systems place ad hoc restrictions on what modifications can be made through views, but quest for better solutions is ongoing; see [389, 390] for overview and [238, 327] for recent developments.

This section focuses on the first maintenance setting described above — maintaining views in response to base data modifications. As illustrated in Example 1.3, the single most important idea in view maintenance is perhaps *incremental view maintenance*, which builds on the observation that computing changes to view contents induced by small changes to base tables may be more efficient than recomputing views from large base tables. Figure 2.1 illustrates incremental maintenance versus full recomputation of a materialized view. Much of the discussion in this section is about how to make incremental maintenance more efficient, flexible, and general, and how to choose view maintenance strategies (including recomputation) intelligently.

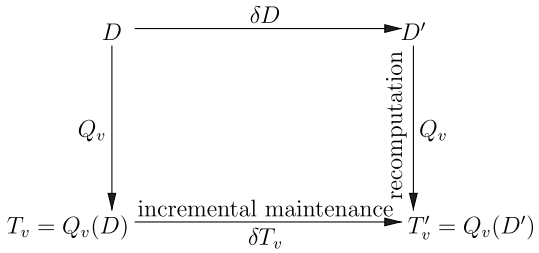


Fig. 2.1 Incremental view maintenance versus recomputation of materialized view (Q_v, T_v).

Besides the potential performance advantage, it is interesting to note that incremental maintenance, or, more generally, *incremental query computation*, also adds to the expressive power of a query language [141]. For example, it is well known that the transitive closure query cannot be expressed in relational algebra, but it can be expressed in an incremental setting.

Example 2.1. Consider a directed graph represented by a table `edge(v1,v2)`, where each row corresponds to a directed edge from vertex `v1` to `v2`. To compute the transitive closure `reachable(v1,v2)` of the graph, we start with an empty `reachable`. We process each row of `edge` as an insertion into `edge`, and maintain `reachable` as follows:

$$\texttt{reachable} \leftarrow \texttt{reachable} \\ \cup \Delta\texttt{edge} \cup \pi_{\texttt{v1,v3}}(\Delta\texttt{edge} \bowtie \rho_{(\texttt{v2,v3})}\texttt{reachable}).$$

Here, Δ`edge(v1,v2)` denotes a table containing the (single) row inserted into `edge`, ρ renames the columns in a table, and $\cup$, π, and $\bowtie$ respectively denote relational (and hence duplicate-eliminating) union, projection, and natural join. Using the same rule above, we can also maintain `reachable` as a materialized view with respect to single-row insertions into `edge`.

Section Roadmap We now provide a roadmap for our discussion of view maintenance in this section. The survey by Gupta and Mumick [187] and the overview by Deligiannakis [132] classified work on view maintenance along multiple dimensions. We highlight three of these dimensions here: (1) *language* (and data model) used in defining views (e.g., relational or XML), (2) *information* used by the maintenance procedure (e.g., whether base tables are accessible), and (3) *timing* of maintenance (e.g., whether a view is maintained immediately upon base table modifications or upon the first time it is queried).[1]

[1] Gupta and Mumick's classification in [187] included also *allowable modification* (what modifications the algorithm can handle) and *database instance* (whether the algorithm works for all or only some of the database instances) as dimensions, but did not include

An exhaustive review of all techniques along with the language dimension is beyond this scope of this survey, so we focus on nonrecursive SQL views. The remainder of this section discusses techniques along with the information and timing dimensions, as well as issues that do not fit the classification above:

- Section 2.1 begins with a discussion of how incremental view maintenance is algorithmized and implemented, laying the foundation for subsequent sections. We start with the *algebraic* (as opposed to procedural) approach toward specifying view maintenance. Although conceptually clean and influential, the approach leaves out many implementation issues. We then highlight *derivation counting* (a technique for maintaining result rows with multiple derivations from base data) in Section 2.1.1, and alternative ways for representing base data changes (which also affect the efficiency of view maintenance) in Section 2.1.2. Finally, in Section 2.1.3, we outline practical issues that arise when implementing view maintenance in a database system, such as how to apply computed changes to materialized views, how to optimize view maintenance (including choosing between recomputation and incremental maintenance), and how to maintain multiple materialized views efficiently.
- Section 2.2 surveys view maintenance techniques that vary along with the *information* dimension. Besides exploiting the knowledge of database constraints, different techniques assume different types of data to be accessible by view maintenance. Section 2.2.1 discusses detection of *irrelevant updates*, which accesses only base table modifications; Section 2.2.2 discusses *self-maintainable* views, whose

information or timing. In [188] they further added *auxiliary information* (what and how much information the algorithm maintains in addition to aid maintenance) and *complexity improvement* (whether the algorithm is demonstrably better than recomputation), and further distinguished *view maintenance language* from view definition language. Deligiannakis's classification [132] included — in addition to language, information, and timing — the dimensions of allowable modifications and *algorithm applicability* (analogous to the database instance dimension in [187]).

maintenance accesses base table modifications and view contents, but *never* base tables; Section 2.2.3 discusses *runtime self-maintenance*, which does not preclude access to base tables, but tries to avoid that as much as possible on a pre-modification basis. By limiting their access to base data, these approaches can be more efficient and reliable, particularly in distributed settings like a data warehouse.

- Section 2.3 discusses what is actually being materialized — there are more interesting options than simply storing the view contents in a table. Section 2.3.1 shows how, instead of literally materializing the view contents, storing a data structure in place of the contents may save space and facilitate queries and maintenance. Section 2.3.2 shows how maintaining more (data in *auxiliary views*) can mean maintaining less (in terms of maintenance costs).
- Section 2.4 examines view maintenance policies that differ in the *timing* dimension. We begin in Section 2.4.1 with the straightforward approach of *immediate* maintenance, which maintains views in the same transaction as modifications to the base tables. Then, in Section 2.4.2, we survey the work on *deferred maintenance*, which is more flexible and benefits from efficient batch processing of modifications.
- Section 2.5 covers other issues, including concurrency control (Section 2.5.1), distributed and parallel processing (Section 2.5.2), and a brief survey of support for view maintenance in database products (Section 2.5.3).

2.1 Algorithmizing and Implementing Maintenance

A popular way of specifying incremental view maintenance algorithms in the literature is the *algebraic* approach, which we will focus on in this section. Examples of nonalgebraic approaches include the *counting algorithm* of Gupta et al. [189], which supports views with bag semantics, group-by, and recursion expressed as Datalog programs, and the method based on production rules by Ceri and Widom [83], which handles SQL views with one level of nesting and one set union or intersect

operator. We choose to focus instead on the algebraic approach because it is more prevalent and easier to explain.

With the "purest" algebraic approach, popularized by Qian and Wiederhold [331], changes to base data are represented as a pair of so-called *delta tables* — ∇R and ΔR — for each base table R. ∇R contains deleted rows and ΔR contain inserted rows; in-place updates to rows are represented as deletion of their old versions in ∇R followed by insertion of their new versions in ΔR. Under the set semantics (i.e., no table contains duplicate rows), the net effect on R is $R \leftarrow R - \nabla R \cup \Delta R$. Under the bag semantics (i.e., tables are bags of rows with possible duplicates, which is the default in SQL), the net effect on R is $R \leftarrow R \dot{-} \nabla R \uplus \Delta R$, where $\dot{-}$ and $\uplus$ denote bag difference and union, respectively, to distinguish them from their relational counterparts $-$ and $\cup$. In general, one transaction may modify multiple tables.

With the algebraic approach, we specify view maintenance using a collection of *change propagation equations*. Each equation specifies, for an operator of the view definition language, how to "propagate" changes from one input table. At compile time, given a view definition query Q_v expressed as an operator tree, we apply the change propagation equations to the tree in a bottom-up fashion. Roughly speaking, starting with changes to base tables, we derive queries for computing changes to progressively larger subexpressions of Q_v, and eventually obtain queries for computing changes to Q_v. At run time, given the changes to base tables, we evaluate these queries to obtain changes to the view contents, and then "refresh" the view by applying these changes. The example below illustrates some change propagation equations and their application.

Example 2.2. The change propagation equations for selection (σ_p) and (theta-)join ($\bowtie_\theta$) in bag algebra are as follows (they also work under set semantics if operators are replaced by their set counterparts).

$$\sigma_p(R \dot{-} \nabla R) = \sigma_p(R) \dot{-} \sigma_p(\nabla R) \tag{2.1}$$

$$\sigma_p(R \uplus \Delta R) = \sigma_p(R) \uplus \sigma_p(\Delta R) \tag{2.2}$$

$$(R \dot{-} \nabla R) \bowtie_\theta S = (R \bowtie_\theta S) \dot{-} (\nabla R \bowtie_\theta S) \tag{2.3}$$

$$(R \uplus \Delta R) \bowtie_\theta S = (R \bowtie_\theta S) \uplus (\Delta R \bowtie_\theta S) \tag{2.4}$$

These equations all have the same form: the left-hand side of each shows a single operator with one of its input changed (by either ∇R or ΔR); and the right-hand side begins with the same operator with inputs unchanged, followed by changes to the result of this operator.

Using the equations above, we show how to derive changes to the view $Q_v = \sigma_p(R) \bowtie_\theta S$, given ∇R, ΔR, and ΔS:

$$
\begin{aligned}
&\sigma_p(R \dot{-} \nabla R \uplus \Delta R) \bowtie_\theta (S \uplus \Delta S) \\
&\quad = \big(\sigma_p(R) \dot{-} \sigma_p(\nabla R) \uplus \sigma_p(\Delta R)\big) \bowtie_\theta (S \uplus \Delta S) && \text{by (2.2), (2.1)} \\
&\quad = \big(\sigma_p(R) \bowtie_\theta (S \uplus \Delta S)\big) \\
&\qquad \dot{-} \big(\sigma_p(\nabla R) \bowtie_\theta (S \uplus \Delta S)\big) \\
&\qquad \uplus \big(\sigma_p(\Delta R) \bowtie_\theta (S \uplus \Delta S)\big) && \text{by (2.4), (2.3)} \\
&\quad = (\sigma_p(R) \bowtie_\theta S) \uplus (\sigma_p(R) \bowtie_\theta \Delta S) \\
&\qquad \dot{-} (\sigma_p(\nabla R) \bowtie_\theta S) \dot{-} (\sigma_p(\nabla R) \bowtie_\theta \Delta S) \\
&\qquad \uplus (\sigma_p(\Delta R) \bowtie_\theta S) \uplus (\sigma_p(\Delta R) \bowtie_\theta \Delta S) && \text{by (2.4).}
\end{aligned}
$$

There are several subtleties. First, if we want to represent the changes derived above by a pair of delta tables, some care is required. It would be wrong to combine all negative change terms — $\sigma_p(\nabla R) \bowtie_\theta S$ and $\sigma_p(\nabla R) \bowtie_\theta \Delta S$ — into ∇T_v and all positive change terms — $\sigma_p(R) \bowtie_\theta \Delta S$, $\sigma_p(\Delta R) \bowtie_\theta S$, and $\sigma_p(\Delta R) \bowtie_\theta \Delta S$ — into ΔT_v, and simply modify the materialized view contents T_v with $T_v \leftarrow T_v \dot{-} \nabla T_v \uplus \Delta T_v$. The reason is that deleting $\sigma_p(\nabla R) \bowtie_\theta \Delta S$ before inserting $\sigma_p(R) \bowtie_\theta \Delta S$ may not achieve the intended effect because of the proper subtraction semantics of $\dot{-}$. An excellent reference on how to handle this problem (and other subtleties) is [180].

Second, note the change terms derived above are defined using the state of base tables R and S *before* they are changed. Depending on its timing, a view maintenance procedure may see newer states of base tables and therefore needs to compute changes differently from above. Section 2.4 will have more to say about this issue.

Finally, the above derivation handles multiple base table deltas *simultaneously*. Performing view maintenance one delta table at a time often results in much simpler expressions. For example, T_v can also be maintained as follows together with the base tables, by processing each

of ∇R, ΔR, and ΔR in order:

$$T_v \leftarrow T_v \dot{-} (\sigma_p(\nabla R) \ltimes_\theta S); \qquad R \leftarrow R \dot{-} \nabla R;$$
$$T_v \leftarrow T_v \uplus (\sigma_p(\Delta R) \ltimes_\theta S); \qquad R \leftarrow R \uplus \Delta R;$$
$$T_v \leftarrow T_v \uplus (R \ltimes_\theta \Delta S); \qquad S \leftarrow S \uplus \Delta S.$$

The algebraic approach toward specifying incremental view maintenance has been influential. Blakeley et al. [50] gave maintenance expressions for relational select–project–join views. Qian and Wiederhold [331] proposed change propagation equations for relational algebra operators. This approach was perfected by Griffin and colleagues for bag algebra [180] (including a final aggregation function but no group-by) and relational algebra [181], and extended with semijoins and outerjoins [179]. Quass extended the approach to bag algebra plus the group-by-aggregation operator [332]. Mumick, Quass, and Mumick considered SQL group-by-aggregation views (whose definitions had aggregation as the last operator) [307], and proposed an alternative method of representing changes called *summary-deltas*, which we shall discuss further below. Gupta and Mumick [197] generalized the notion of summary-deltas and developed an algebraic approach that handled bag algebra plus group-by-aggregation and outerjoin operators. The latest on maintaining outerjoin views (possibly followed by aggregation) is by Larson and Zhou [262].

The algebraic approach is attractive for several reasons. Besides being conceptually simple, it is modular and composable: change propagation equations are defined separately for each operator in the view definition language, and together they can handle arbitrarily complex query expressions. In its purest form, this approach expresses all maintenance tasks using only the standard query operators, and hence can leverage existing query execution and optimization engines. However, in reality, as we shall see below, efficient implementation sometimes requires giving up the conceptual simplicity of the algebraic approach and specifying aspects of the maintenance in a more procedural manner. Therefore, the division between algebraic and procedural approaches toward specifying incremental view maintenance has been increasingly

blurred. For example, summary-deltas and variants (Section 2.1.2), which have emerged as the preferred method of representing changes, require new operators or special procedures to apply.

2.1.1 Derivation Counting

A well-known difficulty in incremental view maintenance is handling projection in relational algebra or duplicate elimination in bag algebra. Consider a view $\pi_A(R)$ in relational algebra, which contains the set of distinct A values in R. Suppose we delete a row r from R. To maintain the view, we must know how many other rows in R have the same A value as r: if there is none, then we should delete $r.A$ from the view; otherwise, the view remains unchanged. This check requires examining R, which can be expensive in general, e.g., when R itself is a complex query expression instead of a base table.

A key idea for coping with this difficulty is *derivation counting*. That is, for each materialized row, we also keep track of the count of its derivations. For example, for each row (A value) in the materialized $\pi_A(R)$, we also store the number of rows in R that contributes to this result; i.e., those with the same A value. This count can be easily maintained when R is modified. If a deletion from R causes the count associated with a row in $\pi_A(R)$ to become 0, then this row is deleted. This idea of counting has been used for relational view maintenance since [50], and also underpinned the *counting algorithm* of Gupta et al. [189]. Counting is useful not only for maintaining views involving projection and duplicate elimination, but also group-by-aggregation, as we will see later in Example 2.5.

Note that derivation counting requires extending the algebraic approach. The basic change propagation equations can be inefficient if applied directly, as they do not consider or use extra information such as counts. For example, the equation for propagating deletion through relational projection [331] is

$$\pi_A(R - \nabla R) = \pi_A(R) - (\pi_A(R) - \pi_A(R - \nabla R)),$$

which implies recomputing the view. In this case, we must rely on other techniques (Section 2.3) for deciding what to materialize — in

addition to or instead of the view itself — to make view maintenance more efficient. In this sense, the algebraic approach may not be as "complete" a solution as a procedural one exemplified by [189].

2.1.2 Alternative Representations of Changes

So far, we have been assuming — as most early work on the algebraic approach did — that changes to a table R are represented as delta tables ∇R and ΔR. This representation is conceptually appealing, because delta tables have the same schema as their corresponding tables, and can be applied using operators in the view definition language. However, this representation turns out to be inefficient in some situations.

Example 2.3 (Adapted from [197]). Recall Example 1.1. Suppose we materialize a view defined by `(Q1v)`, but not `TotalByItemStore`. Consider the insertion of five rows, represented by Δ`pos`, into base table `pos` as in Example 1.3. If we maintain `(Q1v)` by first propagating changes through `TotalByItemStore`, and if we represent the changes to `TotalByItemStore` using a pair of delta tables ∇`TotalByItemStore` and Δ`TotalByItemStore`, then we have to determine the old `total` for each row in ∇`TotalByItemStore`, which requires recomputing it from the corresponding group from `pos` because `TotalByItemStore` is not materialized. A better strategy would to be represent only the net effect on `TotalByItemStore`, which in this case consists of five rows, each representing a value to be added to the `total` of an affected `TotalByItemStore` row. This strategy eliminates the need to obtain the actual `total` for each affected row, but we need new equations for propagating this "add-to" style of changes further through other operators in the view definition expression.

Example 2.4. Consider a view $\top_A^k(R)$, which returns the top k rows in R as ranked by their A values (for simplicity, ignore the case of ties). One of the top k rows in R, say r, is updated, such that its updated version, r', continues to remain in the top k. Suppose this update is

represented as a pair of delta tables, $\nabla R = \{r\}$ and $\Delta R = \{r'\}$, and propagated in order. When propagating ∇R, we need to compute the $(k+1)$-th ranked row in R because r is deleted. However, when propagating ΔR, we will simply discard this row because of the insertion of r'. It would be more efficient to detect that the update does not actually change the membership of the top k rows and hence avoid querying R for the $(k+1)$-th ranked row. Doing so would require that we do not "break up" an update into a pair of deletion and insertion.

Inefficiency associated with the delta tables has led to the development of alternative representations of changes that capture the net effect of changes more precisely and in a minimalistic manner. Mumick et al. [307] proposed computing and representing changes to an aggregate view as a *summary-delta* that encodes the net effect of insertions, deletions, and updates in a single table, as illustrated by the following example.

Example 2.5. Consider the following materialized view, modified from `TotalByItemStore` (Example 1.1) by adding a column for derivation count. For simplicity, assume that `qty` and `price` cannot contain `NULL` values.

```
CREATE VIEW ISTC(itemID, storeID, total, count) AS
  SELECT itemID, storeID, SUM(qty*price), COUNT(*)
  FROM pos GROUP BY itemID, storeID;
```

Inserting row $\hat{r}$ into `pos` would generate, for `ISTC`, a summary-delta row $\langle \hat{r}.\texttt{itemID}, \hat{r}.\texttt{storeID}, (\hat{r}.\texttt{qty} \times \hat{r}.\texttt{price}), 1\rangle$, while deleting $\check{r}$ from `pos` would generate $\langle \check{r}.\texttt{itemID}, \check{r}.\texttt{storeID}, -(\check{r}.\texttt{qty} \times \check{r}.\texttt{price}), -1\rangle$.

Now, consider a row r in the summary-delta for `ISTC`. If `ISTC` currently contains no row with r.`itemID` and r.`storeID`, we insert r into `ISTC`. Otherwise, we update the row in `ISTC` with r.`itemID` and r.`storeID` by adding r.`total` to its `total` and r.`count` to its `count`. If the resulting `count` becomes 0, we delete the row from `ISTC`.

Because of their efficiency advantage over delta tables, summary-deltas have become the representation of choice especially for views

involving aggregation [253, 307, 314]. Some commercial database systems use summary-deltas (or their variants) in providing built-in support for materialized views, e.g., [43, 262, 269, 320]. Gupta and Mumick [197] generalized the notion of summary-deltas to *change tables*, and developed an algebraic framework that allows change tables to be propagated through complex view definition expressions; a special "refresh" operator was introduced to apply a change table.

Yet another way to represent changes is to use the SQL statements that produced them. In SQL, rows to be deleted and updated are selected by a `WHERE` condition. Reasoning with these modification conditions and the conditions in the view definition queries sometimes allows us to simplify maintenance. Blakeley et al. [48, 49] took this approach in detecting changes whose effect on a view could be computed by just examining the changes and the view definition; more on detecting such changes will be discussed in Section 2.2.

2.1.3 Implementation and Optimization

We now outline several practical issues that arise when implementing view maintenance in a database system.

Applying Computed Changes to Materialized Views One practical problem with implementing incremental maintenance is that there has been no convenient way to express bag deletion in SQL,[2] which is needed for applying computed delta tables to materialized views. It is not until recently that some database systems have introduced extensions to `DELETE` that allow bag deletion to be expressed more directly. In general, it has been observed [253, 314] that most database systems have a difficult time with modification statements that arise in view maintenance; efficient application of summary-deltas requires implementation using stored procedures with cursors.

Optimizing View Maintenance In an algebraic approach, it is difficult to measure or ensure the efficiency of change propagation

[2] Bag difference, as a query operator, is supported by `EXCEPT ALL`, but it does not lead to an efficient implementation of bag deletion.

equations and the change expressions they generate. The *minimality* requirement was proposed by [180, 331]: if changes to view V are calculated as delta tables ∇V and ΔV, then we require $\nabla V \subseteq V$ and $\nabla V \cap \Delta V = \emptyset$. Intuitively, this condition ensures that the changes we derive are minimal. However, as pointed out by [180, 331], minimality can be trivially satisfied by letting $\nabla V = Q_v \dot{-} Q'_v$ and $\Delta V = Q'_v \dot{-} Q_v$, where Q'_v is obtained from Q_v by replacing references to base tables with subexpressions computing their new states (just like the left-hand sides of change propagation equations in Example 2.2). However, simply using the above definitions to maintain V is even costlier than recomputing V. Ultimately, their efficiency should be vetted by a cost-based database optimizer.

In general, it would be best to leave the decision on how to maintain a view to the database optimizer, because the optimal strategy depends not only on the type of the view but also on the workload and other information available only at the time of maintenance. As a simple example, consider the basic question of whether incremental maintenance is always better than recomputation. Not surprisingly, the answer is no; recomputation may well win for certain views and workloads, e.g., when significant portions of the base tables have changed [51, 120].

Traditional database optimizers must be extended to handle view maintenance. For example, an optimizer can use the algebraic approach to generate change propagation equations, further optimize them as queries, and compare the estimated overall maintenance cost with that of recomputation. Vista [392, 393] considered how to extend a transformation-based optimizer to handle view maintenance queries. In particular, knowledge of constraints (such as keys and foreign keys) can be used by the optimizer in simplifying maintenance queries. Bunger et al. [65] used dynamic maintenance plans that could make run-time decisions of strategies (incremental maintenance versus recomputation) based on the actual intermediate results produced by maintenance.

Maintaining Multiple Views A database with multiple materialized views needs to carry out multiple maintenance tasks when the base tables change. A materialized view defined in terms of other materialized views can often be maintained more efficiently by exploiting the

availability of these other views. Furthermore, similar subexpressions within and across maintenance queries can be identified, and their evaluation can be shared instead of repeated. Segev et al. [351, 352, 353] explored how to share the work of maintaining multiple select–project views defined over the same base table. Labio et al. [254] studied, for a directed acyclic graph of views defined over base tables and other views, how to coordinate computation and application of changes in order to minimize maintenance cost. They also considered the possibility of simplifying maintenance by processing changes for one base table or view at time (instead of simultaneously) in a particular order, as illustrated at the end of Example 2.2. Mistry et al. [305] applied multi-query optimization techniques to optimize multiple view maintenance queries. Lehner et al. [268] considered merging similar subexpressions across maintenance queries into a common, subsuming expression whose result could be shared; their approach was subsequently extended by Xu et al. [402] to handle a very large number of views. DeHaan et al. [131] discussed the maintenance of materialized views defined over other views. Folkert et al. [151] studied how to schedule maintenance of multiple views such that already maintained views could be used to support maintenance of others. Recent work on exploiting similar subexpressions for query processing by Zhou et al. [416] considered the optimization of multiple view maintenance queries as one of the main applications of their techniques.

Discussion Broadly speaking, the problem of rewriting a view maintenance query to use the contents of this or other materialized views is an instance of the problem of answering queries using views (Section 3). The problem of selecting additional views to materialize for the purpose of speeding up view maintenance — further discussed in Section 2.3 — is an instance of the view selection problem (Section 4). To sum up, efficient realization of view maintenance requires many techniques beyond simple application of the algebraic approach.

2.2 Information Available to Maintenance

Change propagation equations presented earlier in Section 2.1 are rather generic and only reference base and delta tables. In practice,

maintenance can make use of a variety of alternative and additional information, such as the materialized view contents. knowledge of database constraints (such as keys and referential integrity constraints). The following example illustrates this point.

Example 2.6. Consider the `items` table from Example 1.1 and a materialized view V defined by $\pi_{\texttt{name}}(\sigma_{\texttt{cost}>100}\texttt{items})$, which returns the distinct (by name) items that are "expensive." Suppose a new row $\hat{r}$ is inserted into `items`. We can maintain V as follows.

- If `name` is a key of `items`, we insert $\hat{r}.\texttt{name}$ into V if and only if $\hat{r}.\texttt{cost} > 100$.
- If `name` is not a key (or we do not know that it is), we must additionally verify that V does not already contain $\hat{r}.\texttt{name}$ before inserting it into V.

Now suppose an existing row $\check{r}$ is deleted from `items`.

- If `name` is a key of `items`, we delete $\check{r}.\texttt{name}$ from V if and only if $\check{r}.\texttt{cost} > 100$.
- If `name` is not a key (or we do not know it is), we must check the base table to see how many remaining expensive items bear the same name; we delete $\hat{r}.\texttt{name}$ from only if none remains.

As we can see from above, view maintenance may require access to different amounts of information. Sometimes the deltas alone suffice, while sometimes the materialized view contents or even the base tables are needed. Constraints can simplify view maintenance by reducing the amount of data that it needs.

There are many scenarios where performing view maintenance with partial access to data is beneficial, for both efficiency and feasibility reasons. From deltas to materialized views to base tables, we will likely find increasing sizes and access costs. For example, consider a scenario where a central data warehouse maintains a materialized view over base tables stored at remote data sources. Suppose the sources push deltas

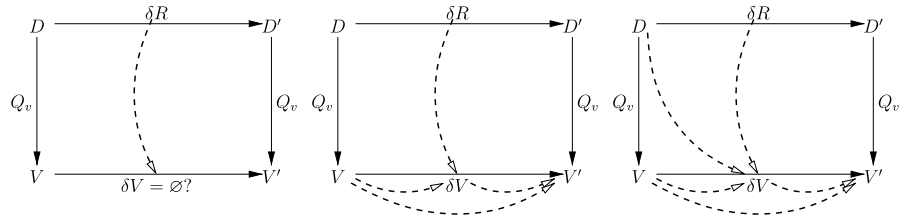
to the warehouse. It would be nice for the warehouse to be able to maintain its view using just the deltas and the view contents stored locally, without performing expensive queries over remote, and possibly unavailable, base tables. Furthermore, if the data sources can use the knowledge of constraints and the view definition to determine that certain deltas will not affect the warehouse view, we can save communication and further processing by not sending such deltas to the warehouse.

In the remainder of this section, we discuss several concepts and results on maintaining views using partial information. They vary in the amount of information required and offer a spectrum of trade-offs between efficiency and applicability — those that require less data may be more efficient, but they work for fewer views and modifications.

2.2.1 Irrelevant Updates

Given a base table modification δ, sometimes we can determine that it has no effect on the contents of a view V for all possible database states. Such modifications were termed *irrelevant updates* by Blakeley et al. [48, 49, 50]. As a simple example, consider the insertion of a new row into the `items` table with `cost` $= 99$, which does not satisfy the selection condition of view V in Example 2.6. It is clear that this insertion is irrelevant to V.

The idea of irrelevant updates can be traced back to the related context of supporting data triggers and alerters, where they were called *readily ignorable updates* by Buneman and Clemons [64]. Blakeley et al. [48, 49] showed how to detect irrelevant updates by testing satisfiability of Boolean expressions constructed from the view definitions and modification specifications (as mentioned earlier in Section 2.1.2, deletions and updates are represented in [48, 49] using conditions that select the rows they apply to). Such tests do not access the contents of base tables or materialized views. More generally, for recursive views (defined in Datalog), the problem of detecting irrelevant updates can be reduced to that of testing equivalence of Datalog programs [273]: though undecidable in general, algorithms for restricted subclasses have been developed [144, 273].



(a) Irrelevant update dectection (b) Self-maintenance (c) Unrestricted maintenance

Fig. 2.2 Comparison of the types of data accessed by irrelevant update detection, self-maintenance, and unrestricted maintenance. The dashed arcs show what types of data are involved in computation.

2.2.2 Self-Maintainable Views

A view V is *self-maintainable* with respect to a modification type (insertion, deletion, or update) to a base table R if, for all possible database states and modifications δ to R of this type, we can *self-maintain* V, i.e., using its materialized contents and δ but no base tables. Figure 2.2 illustrates the types of data accessed by self-maintenance, in comparison with irrelevant update detection (Section 2.2.1) and unrestricted maintenance.

Example 2.7. Consider a view defined by `(Q1)` from Example 1.1. Since `pos(itemID)` is a foreign-key reference to `items(itemID)`, this view is self-maintainable with respect to insertions into `items`: a new `items` row cannot possibly join with existing `pos` rows and therefore cannot affect the view. However, if there is no referential integrity constraint from `pos(itemID)` to `items(itemID)`, the view will not be self-maintainable with respect to insertions into `items`, because we need to access base table `pos` to find rows that join with the new items.

The view is not self-maintainable with respect to insertions into `pos` because new `pos` rows need to be joined with base table `items` to find the categories of items sold.

The notion of self-maintainability generalizes to multiple views. A set $\mathcal{V}$ of views is self-maintainable with respect to a modification type to a base table if, for all such modifications and possible database states,

we can self-maintain all views in $\mathcal{V}$, i.e., using their materialized contents and the modifications but no base tables.

Example 2.8. Consider a set of two views: V_1 is defined by `(Q1)` from Example 1.1 as in Example 2.7, and V_2 is defined as follows:

```
SELECT itemID, category FROM items;
```

Assuming reasonable key and foreign-key constraints, $\{V_1, V_2\}$ is self-maintainable with respect to insertions into both `items` and `pos`. The contents of V_2 can be used to maintain V_1 upon insertions into `pos`. V_2 is in fact self-maintainable by itself.

Deletions from `pos` are tricky. It may appear that we can simply join a deleted `pos` row $\check{r}$ with materialized V_2 to get the item's category, and maintain V_1 by decrementing the appropriate sum by $\check{r}$`.qty*`$\check{r}$`.price`. However, if $\check{r}$ represents the only sale in its category at its store, we should remove $\check{r}$'s corresponding group from V_1. If there are database constraints requiring both `pos.qty` and `pos.price` to be non-`NULL` and positive, we can handle this case by checking whether the sum drops to 0, so $\{V_1, V_2\}$ is self-maintainable with respect to deletions from `pos`. Without these constraints, however, V_1 must also have `COUNT(*)` and `COUNT(qty*price)` in its `SELECT` clause for $\{V_1, V_2\}$ to be self-maintainable.[3]

Compared with the notion of irrelevancy discussed above, which is a property of one specific modification, self-maintainability — as defined here and by Gupta et al. [186] — is a property of a view or set of views, and it implies the ability to maintain view(s) without accessing base tables for all possible modifications. This powerful guarantee makes self-maintainable views especially attractive to data warehousing, because they completely avoid the need to access remote base tables, which may be expensive and face consistency issues with asynchronous base table modifications (Section 2.5.1).

[3] If a group's `COUNT(*)` drops to 0, it needs to be removed; otherwise, if its `COUNT(qty*price)` drops to 0, its sum needs to change from 0 to `NULL`.

Gupta et al. [186] showed how to test whether a select–project–join view (including self-join and outerjoin) is self-maintainable, and if yes, how to self-maintain it. Unfortunately, most views are not self-maintainable. Follow-up works [29, 217, 278, 306, 333, 347] address the closely related problem of making a view self-maintainable if it is not; they also extended self-maintainability and self-maintenance results to consider semijoins, aggregates, as well as multiple views and additional database constraints. We defer their discussion to Section 2.3, because making a view self-maintainable entails maintaining auxiliary data.

2.2.3 Run-Time Self-Maintenance

Another line of work on self-maintenance takes an alternative approach: instead of requiring self-maintainability for all possible modifications of a given type, we examine self-maintainability on a premodification basis at run time. A view is *(run-time) self-maintainable* with respect to a base table modification δ if we can self-maintain the view using its contents and δ. Compared with the stricter, "compile-time" notion of self-maintainability in Section 2.2.2, we no longer guarantee self-maintainability at all times; however, the hope is that we can still avoid accessing the base tables for many modifications, thereby improving the average maintenance cost.

Example 2.9. Consider the set $\{V_1, V_2\}$ from Example 2.8. As discussed, this set is not self-maintainable with respect to deletions from `pos` in general (e.g., when `price` may be 0), because of the special case when a deletion causes a group's sum to drop to 0. However, given a particular deletion of row $\check{r}$ from `pos`, we can test whether the current sum for $\check{r}$'s group materialized in V_1 is strictly greater than $\check{r}$`.qty*`$\check{r}$`.price`. If yes, V_1 is self-maintainable with respect to $\check{r}$, and we simply decrement the sum by $\check{r}$`.qty*`$\check{r}$`.price`. Otherwise, V_1 is not self-maintainable with respect to $\check{r}$.

Example 2.10 (Adapted from [48]). Consider base tables $R_1(H, I)$ and $R_2(J, K)$ and a view V defined as $\pi_{J,K}(\sigma_{H<20} R_1 \bowtie_{I=J} R_2)$ under the set semantics. Suppose a modification δ deletes all rows in R_1 with

$I = 20$ and $H < 30$. V is self-maintainable with respect to δ; to maintain V we simply delete all its rows with $J = 20$. The justification is that all such rows must have been derived from R_1 rows with $I = 20$ and $H < 20$, which are deleted by the modification because $H < 20$ implies $H < 30$.

On the other hand, consider the deletion of row $\langle 10, 20 \rangle$ from R_1. V is not self-maintainable with respect to this deletion. There may be any number of other R_1 rows with $I = 20$ and $H < 30$; therefore, without checking R_1, we do not know whether to delete V rows with $J = 20$.

The idea of run-time self-maintenance was first explored by Blakeley et al. [48, 49] (and was termed *autonomously computable updates*). They developed tests for run-time self-maintainability and procedures for run-time self-maintenance of select–project–join views, though they did not exploit the currently materialized view contents. Tompa and Blakeley [382] extended the approach to make use of view contents. Gupta and Blakeley [184] further improved the approach, and also considered the case when some but all of the base tables can be accessed for view maintenance (which can no longer be classified as self-maintenance). Huyn [220] proposed more efficient tests for run-time self-maintenance by separating compile-time test generation from run-time test evaluation. In [221], Huyn tackled run-time self-maintenance of multiple views, handled multiple insertions and deletions in a batch, and considered the case when some base tables may be available; one key insight was that self-maintainability testing can be reduced to query containment.

Tests of run-time self-maintainability and irrelevancy (when using view contents) are hard decision problems; even for select–project–join views they are co-NP-complete in the size of the view contents. To make them practical, Huyn developed efficient (polynomial in the size of the view definition) tests for subclasses of select–project–join views with no self-joins or a limited form of them [222].

2.3 Materialization Strategies

Suppose we have decided to materialize a view, for example, to speed up user queries. The simplest strategy is to store it (and it alone)

as a data base table. More sophisticated strategies are often needed, however. We have already seen examples where materializing additional view subexpressions (Sections 2.1.3 and 2.2.2) and information such as counts (Sections 2.1.1 and 2.2.2) can improve view maintenance, even though they do not directly improve user queries. Moreover, instead of materializing a view as a table, we can store it using data structures that can be accessed and/or maintained more efficiently, much like indexes. We devote this section to discussing strategies that materialize alternative structures or additional information.

2.3.1 Alternatives to Materializing Contents

Much of the work on materialized views aims at using their contents to speed up queries. While this "logical" approach to reuse is attractive for its simplicity and physical data independence, a more "physical" approach, which reuses both data and access paths, can potentially offer higher efficiency. A representative of this approach is the *ADMS* project of Roussopoulos et al. [341, 340, 342]. Instead of materializing each actual result row of a select–join view, *ADMS* uses a *ViewCache* to store pointers to base table rows (or entries in other *ViewCaches* corresponding to subexpressions) that contributed to the result row. *ViewCache*'s restricted query form and compact representation make it very efficient to store, maintain, and access. Blakeley and Martin [51] analytically showed workloads where join indexes (which can be seen as *ViewCaches*) outperformed both virtual join views and those materialized as tables.

For complex views, directly materializing their contents is sometimes impractical. We give two specific examples below.

- The *data cube* [172] for a d-dimensional table (i.e., d possible group-by columns) is the union of 2^d group-by-aggregation queries, each with a different subset of group-by columns. Simply materializing this union view as a table is inefficient, as its size is exponential in d. More compact representations are possible by avoiding the redundancies in the summary data. For example, consider again the schema

from Example 1.1. Suppose that a particular item i is sold exclusively through store s. Then, the total venue for i (in the result of grouping `pos` by `itemID`) must be the same as that for (i, s) (in the result of grouping by $\{\texttt{itemID}, \texttt{storeID}\}$), and need not be stored redundantly. A series of work [256, 361, 362, 395] has developed compact storage structures for data cubes to avoid such redundancies. Both Sismanis et al. [361] and Lakshmanan et al. [256] also considered incremental maintenance of their structures. Sismanis et al. [361] showed how to avoid such redundancies when computing the cube in the first place; Sismanis and Roussopoulos [362] further proved that, for uniform base data, both the size and construction cost of their structure were polynomial in d.

- While the above example is motivated primarily by size and construction cost, the next example is motivated by maintenance cost. Consider a table R with a pair of columns `TS` and `TE` that together record, for each row r, the time interval $[r.\texttt{TS}, r.\texttt{TE})$ during which r is "valid" (e.g., an item was being produced during 1992–1996). Suppose we want to materialize a temporal aggregate view $V(\texttt{CNT}, \texttt{TS}, \texttt{TE})$ over R, which counts the number of R rows valid at each point in time, and stores with each count the maximal interval $[\texttt{TS}, \texttt{TE})$ for which it is correct. Materializing V directly makes maintaining V expensive. Inserting a new row $\hat{r}$ into R affects all counts in V whose intervals overlap with $\hat{r}$'s. If $\hat{r}$ has a long valid interval, many rows in V must be updated, so the worst-case cost is linear in $|V|$. Instead, Yang and Widom [404, 405] proposed maintaining a B+tree sorted in the time dimension, whose nodes are augmented with aggregate values (of subtrees). This structure enables V to be maintained in time logarithmic instead of linear in $|V|$ and supports efficient querying and reconstruction of V.

For all work discussed in this subsection, we could have called the materialized data structures indexes instead of views. Indeed, the

distinction between materialized view and indexes has become increasingly blurred — a point that we shall come back to in Section 5.

2.3.2 Maintaining Auxiliary Data

Given a view V to materialize, we may find that materializing appropriate auxiliary data in addition to V makes V more efficient to maintain. Keep in mind that such auxiliary data must be maintained as well. At a first glance, maintaining more data to reduce maintenance costs may appear counterintuitive. However, the idea should not be surprising once we consider an analogy where appropriate indexing — which is another form of auxiliary data — can facilitate base table modifications.

Making Views Self-Maintainable Much of the work on maintaining auxiliary data is aimed at making views (together with the auxiliary data) self-maintainable. Hull and Zhou [217] made a select–project–join view self-maintainable by pushing selections and projections down to base tables and maintaining these select–project views as auxiliary data. To reduce the size of auxiliary data, Quass et al. [333] exploited key and referential integrity constraints and introduced auxiliary views defined using semijoin operators. Akinde et al. [29] extended the approach to views with a final group-by-aggregation following a select join. Mohania and Kambayashi [306] considered views defined by expression trees involving group-by-aggregation as well as set union and difference operators. Going beyond a single given view, Liang et al. [278] and Samtani et al. [347] considered how to make a set of select–project–join views self-maintainable. Garcia-Molina et al. [157] addressed the complementary problem of identifying data to "expire" from a warehouse that were no longer needed for maintaining required information.

Reducing Expected Maintenance Costs As discussed in Section 2.2.2, self-maintainability is a rather strong property; making views self-maintainable for all possible base table modification may require maintaining a lot of auxiliary data and may be expensive. After all, our goal is to lower the cost of maintenance; we can achieve this goal in the expected sense by reducing the frequency of base table accesses instead of completely eliminating them. There has been

some work on using auxiliary data to reduce expected maintenance costs without ensuring self-maintainability. Luo and Yu [300] considered select–join views. Instead of making such a view self-maintainable, they maintained a compact, hash-based data structure for each base table in the view. These structures summarize the values of the join columns for rows passing the local selection conditions. Kept in memory, these summary structures can quickly determine whether a base table modification is relevant to the view; if not, expensive accesses to the corresponding base tables can be avoided.

A series of work considers auxiliary data for `MIN`/`MAX` [320, 399] and, more generally, top-k views (Example 2.4) [406]. A top-k view is not self-maintainable, because when we delete a row currently in the view or lower its value for the ranking column to below that of the currently k-th ranked row, we must access the base table to compute the new k-th ranked row. This base table access can be avoided by maintaining the top k' rows, where $k' \geq k$. Unfortunately, a top-k' is not self-maintainable either. The idea [406] is to adjust auxiliary view definition — k' in this case — as necessary at run time: k' is incremented when an insertion or update causes a row to enter the current top k', and k' is decremented when a deletion or update causes a row to drop out of the current top k'. We query the base table only when k' drops to below k, and in that case, we get more than just the k-th ranked row and reset k' to be greater than k. The gap between the initial k' setting and k is chosen such that it would take some time for k' to become k, allowing the base table access cost to be amortized. As shown by Yi et al. [406], this technique is effective under assumptions that often hold in practice.

Automatic, Cost-Based Selection of Auxiliary Views With techniques for selecting views to materialize (Section 4), an interesting question is whether we can apply them to the workload of view maintenance queries to select appropriate auxiliary views. These techniques will not be able to find views that are approximate summaries (such as in [300]) or whose definitions change dynamically (such as in [406]), but they are attractive because they are automatic and cost-based. Ross et al. [339] pioneered work in this direction. They formalized the

problem of selecting a set of auxiliary views to minimize the total maintenance cost, and proposed a cost-based optimization approach. Labio et al. [252] proposed an approach to this problem based on the A* search, and additionally considered the selection of indexes. A closely related problem is how to select warehouse views to minimize the total query and maintenance costs or under a maintenance cost constraint, which we will further discuss in Section 4.1.1.

2.4 Timing of Maintenance

Different application requirements and workload characteristics call for flexibility in when we carry out view maintenance. In terms of semantics, not all queries need to see the most up-to-date view contents; many can tolerate some staleness, as long as the view contents they access are consistent with some database state not long ago. Even if the queries require up-to-date view contents, we have the option of maintaining a view when its base tables are modified, or when its contents are queried. This section explores view maintenance policies that differ in the timing of maintenance.

2.4.1 Immediate View Maintenance

Among all policies governing the timing of maintenance, the most straightforward is *immediate* view maintenance, which we have assumed implicitly so far: a view is maintained immediately upon any base table modification, as part of the transaction making that modification. This policy implies that view contents are always current; it cannot exploit the applications' tolerance for staleness. While queries benefit from the immediate availability of up-to-date view contents, transactions performing base table modifications must carry the view maintenance overhead. Thus, immediate maintenance is expensive for modification heavy workloads. Furthermore, we cannot use this policy if it is impractical to carry out view maintenance in the same transaction as the base table modifications, e.g., when we maintain warehouse views over remote data sources.

Even with immediate view maintenance, the question remains whether maintenance sees the state of the base tables before or after the

modifications. Most maintenance algorithms discussed in Section 2.1 assume the premodification database state. However, some maintenance algorithms, e.g., [83, 189], make use of both pre- and postmodification states of the base tables. The so-called *state bug* [119] can result if one is not careful when applying such algorithms. A possibility is to compute the premodification base table state from the postmodification one (and vice versa) together with the delta tables. Section 2.4.2 below further discusses how to use the postmodification database state for view maintenance.

2.4.2 Deferred View Maintenance

Overview and Motivation With *deferred* view maintenance, we can modify the view contents to reflect the base table modifications made by a transaction after it commits. This decoupling offers considerable flexibility in the timing of view maintenance. One extreme, which is diametrically opposite to immediate maintenance, is *lazy* (deferred) view maintenance: the contents of a view are brought up to date only when accessed by a query. Transactions modifying base tables are not slowed down, although queries requiring up-to-date view contents must wait for any necessary view maintenance to complete.

Besides lazy, other policies for deferred maintenance are possible. Maintenance can be triggered according to a regular time schedule (called *periodic* or *snapshot* in [188]), after a prescribed number of base table modifications (called *forced delay* in [188]), or in response to some system event or user request. These policies can support various notions and levels of staleness tolerable to applications. They can also be combined with lazy maintenance (i.e., queries always trigger maintenance) to provide up-to-date views to queries.

One key advantage of deferred maintenance is batch processing of modifications. Processing a series of modifications in one batch is generally more efficient than processing them one by one, because it reduces overhead by combining multiple maintenance procedures, allows intermediate states of the view to be skipped, supports condensation of multiple modifications to the same row, and presents more opportunities to optimize maintenance expressions. Batching does incur

the overhead of logging base table modifications for later processing. Although the database recovery log can serve this purpose, extracting modifications for a particular base table and transaction can be expensive. Thus, a more common approach is to log deltas separately for each base table [415, 234].

Literature The idea of deferred maintenance has been around since its application to *database snapshots* [4], though early work in that context [234, 286] dealt only with essentially select–project views, and focused on efficiently identifying relevant base table modifications using storage and logging techniques. *ADMS* [341, 340, 342] was the first system to implement multiple alternative maintenance policies for its *ViewCaches* (see also Section 2.3.1). Hanson [209] analytically compared the performance of immediate and lazy maintenance of materialized views as well as keeping views virtual. Srivastava and Rotem [364] carried out an analytical study of various policies using queuing models, and showed how to pick a policy to minimize a linear combination of average query response time and system processing cost. Segev et al. [351, 352, 353] studied deferred maintenance of multiple select–project views over a remote source; in particular, Segev and Fang [351] considered constraints specifying the maximum view staleness in terms of time. Adelberg et al. [3] studied how to schedule maintenance to balance various factors including the staleness of contents and the response time of queries. Engström et al. [145] developed a framework for selecting warehouse maintenance policies incurring minimum costs in meeting various quality-of-service requirements. An important consideration was the capabilities of data sources, which limit the choice of policies.

Colby et al. [119] provided a formal treatment of deferred maintenance. They specified how to maintain views in bag algebra with the algebraic approach (Section 2.1), using the postmodification database state and avoiding the state bug. To minimize view "downtime" (when views contents are being changed and inaccessible to queries), they proposed separating maintenance into two phases executed in different transactions: *propagate*, which computes changes to views, and *refresh*, which applies the computed changes to views. Only the refresh phase

prevents queries from accessing the views. In [120], Colby et al. studied how to support a combination of immediate, lazy, and periodic maintenance policies for multiple views efficiently. An important subproblem they addressed is ensuring that queries see consistent data across base tables and materialized views they access. Salem et al. [346] proposed maintaining a view in small, asynchronous steps, further reducing the chance of a long propagate phase blocking concurrent base table modifications. Using their algorithm, a view can be brought up to any point in time between the last refresh and the present. The trade-off, however, is that processing deltas in small pieces can be less efficient than processing them in batch [213, 415].

He et al. [213] proposed *asymmetric* batch incremental maintenance for join views. The observation is that asymmetry often exists naturally among different components of the maintenance cost. For example, consider a join view $R \bowtie S$, where R has an index on the join column but S does not. Insertions into S (ΔS) would be cheaper to process than insertions into R (ΔR) and would benefit less from batching. To ensure reasonable query response time, suppose we want to bound the cost of bringing the view up to date whenever its contents are queried. The idea is to process ΔS eagerly (as they arrive) and ΔR as lazy as possible (when the view is queried or the maintenance cost constraint is about to be violated); doing so allows more ΔR to be batched and processed more efficiently. In a follow-up work, Munagala et al. [308] developed a competitive online algorithm for this problem.

Zhou et al. [415] showed how to implement deferred maintenance efficiently in a database system that supports versioning. Maintenance can be triggered by queries or when the system is lightly loaded. Versioning greatly simplifies maintenance expressions because old database states can be readily accessed. Issues like batching, condensing deltas, and recovery are also considered.

2.5 Other Issues of View Maintenance

2.5.1 Concurrency Control

Centralized Database Setting In the context of immediate maintenance (Section 2.4.1), a number of papers specifically address the

concurrency issues that arise in maintaining aggregate views. Transactions that modify different rows in the same group must update the same aggregate value. The standard write locks would cause these transactions to conflict with each other. A key observation is that, for aggregate functions that are associative and commutative (e.g., `SUM`), updates (e.g., increment and decrement) can be applied in any order, and therefore do not conflict with each other. Luo et al. [298, 299] proposed a new locking protocol with a new lock mode exploiting this observation for higher concurrency. While a hash index on the aggregate view was assumed in [299], Luo also studied how to implement the locking protocol with a conventional B-tree index [295]. Instead of creating a new locking protocol, Graefe and Zwilling [170] combined and built on the well-established multiversion concurrency control and escrow locking techniques, and also considered logging and recovery.

Independent of the timing of maintenance, versioning of materialized views is a popular way of improving concurrency between transactions that maintain the views and those that query them. While view maintenance writes to the new version, queries can proceed on an old one. Quass and Widom [334] proposed storing two versions of a table in an extended schema with columns for storing premodification values; queries against original schema were rewritten to use the extended schema. This approach can be generalized to multiple versions using additional columns, as was done by Teschke and Ulbrich [372]. Kulkarni and Mohania [251] considered the alternative of storing versions of a view using separate tables. Kang and Chung [235] developed a multiversion concurrency control mechanism for data cubes materialized as multidimensional array chunks.

For deferred maintenance, we have already seen in Section 2.4.2 many techniques aimed at improving concurrency. The very idea of decoupling view maintenance from base table modifications increases concurrency, and so do separating propagate and refresh phases [119] and breaking up maintenance in small asynchronous steps [346]. However, deferred maintenance complicates consistency because transactions may see stale view contents. While Colby et al. [120] considered consistency between views and base tables, Kawaguchi et al. [240] focused on concurrency control for multiple transactions that read

multiple views, or read a view and also read and/or wrote base tables. The standard strict two-phase locking protocol is no longer enough to ensure serializability in this case.

Distributed Setting Another line of work, started by Zhuge et al. [418], studies concurrency control for view maintenance in a distributed setting, where a data warehouse maintains views over remote base tables. Deferred maintenance is necessary in this case. To maintain a join view in response to a base table modification reported by one data source, the warehouse may need to probe (i.e., query) the other base tables to find joining rows. Meanwhile, additional modifications may have taken place at the sources, so the probes may return answers based on a newer database state. To ensure consistency of the view being maintained, we must compensate for the effects of interfering modifications. Zhuge et al. [418] introduced and solved this problem for the case of a single remote data source. To handle multiple sources, Zhuge et al. [419, 421] proposed the *Strobe* family of algorithms, which decompose maintenance queries into probes against individual sources, keep track of modifications during probe evaluation, and compensate for them later. Building on *Strobe*, the same authors [420] studied how to coordinate maintenance of multiple warehouse views to ensure consistency. Agrawal et al. [18] proposed *SWEEP* algorithms, which compensate each probe individually instead of the whole maintenance query, thereby eliminating the need to wait for quiescence before view refresh while using fewer number of source queries. O'Gorman et al. [313] developed *POSSE*, which balances the use of concurrent source probes (which may yield faster response time by overlapping execution) and sequential ones (which produce smaller intermediate results). Zhang et al. [411, 412] further improved these algorithms by parallelizing the handling of multiple source modifications. Agrawal et al. [17] developed a formal model for maintaining warehouse views by incorporating formalisms from distributed computing. Going beyond data modifications, a series of work by Rundensteiner et al. [100, 102, 104, 287, 288, 413] considered how to handle source schema changes as well. While their early efforts [287, 413] were compensation-based like the algorithms discussed earlier, the work culminated in *TxnWrap* [104]; this multiversion

concurrency control scheme materialized versioned data at the warehouse, sources, or specialized source wrappers.

Note that the idea of maintaining versions of source data is an example of self-maintenance (Section 2.3.2), an alternative solution to the problem of interfering source modifications and probes. Self-maintenance completely avoids this problem by eliminating the need to access base table for maintenance. Nonetheless, there remains the issue of consistency among views dependent on each other for maintenance. We have discussed this issue earlier in this section in the context of deferred maintenance in a centralized database setting, and in Section 2.4 as well. A multiversion approach analogous to [104, 415] offers a possible solution.

2.5.2 Distributed and Parallel Processing

We have already covered some early works on view maintenance in distributed settings in Section 2.4.2, including maintaining remote snapshots [234, 286], *ADMS* [342], and those by Segev et al. [351, 352, 353]. Common to these works is their use of deferred view maintenance because of its natural fit in distributed settings. Another common technique they employ is prescreening base table modifications to eliminate irrelevant ones, thereby reducing communication costs.

In the setting of distributed data warehousing, we have already discussed a lot of work on self-maintenance (Sections 2.2.2 and 2.3.2) and consistency (Section 2.5.1). Additional work in this setting deals with optimization of distributed maintenance queries. When modifications to multiple base tables are present, Liu et al. [290] showed how to restructure the maintenance expression for a view defined as a chain of joins, to reduce the number of source queries. Liu and Rundensteiner [289] further proposed cost-based optimization for distributed maintenance of general join views.

In the setting of parallel database systems, Bamha et al. [39] proposed a parallel maintenance algorithm for select–project–join views under the bulk-synchronous model. Luo et al. [297] considered the task of maintaining join views over horizontally partitioned data. Depending on the table partitioning schemes, a single-node base table modification

may involve all nodes in maintenance, incurring substantial communication. To improve performance, Luo et al. proposed maintaining global indexes or auxiliary views at each node.

In the setting of the emerging *very large scale distributed* (*VLSD*) shared-nothing data storage systems, Agrawal et al. [19] showed how to extend such systems with materialized views using incremental and deferred maintenance. Besides scalability, a key challenge unique to VLSD systems is the stringent fault tolerance requirement, because failures in these systems are much more common than what have been assumed in traditional settings. The solution in [19] adopted a record-level consistency model and built on the asynchronous replication mechanism of VLSD systems.

2.5.3 Implementation in Commercial Database Systems

Materialized views have become a standard feature in most commercial database systems since around the turn of the millennium. The late 1990s and 2000s saw a surge of work from commercial database vendors and their affiliated research labs on materialized views. This section provides pointers to such papers concerning materialized view maintenance (and to sections of this monograph where they are discussed). For pointers to papers discussing view selection and view use in commercial database systems, see Sections 4.2 and 3.2, respectively.

Oracle was among the first to offer support for materialized views in their product. Although the term "materialized views" was first used in Oracle 8i, the feature existed in Oracle 7 and was called "snapshots." View maintenance in Oracle is discussed in [43, 151] (see also Sections 2.1.2 and 2.1.3).

In IBM DB2, materialized views were first called "automatic summary tables" and then "materialized query tables." Their maintenance is discussed in [268, 269, 320, 402] (Sections 2.1.2, 2.1.3, and 2.3.2). View maintenance in Informix (now IBM) Red Brick is described in [65] (Section 2.1.3).

In Microsoft SQL Server, materialized views are also known as "indexed views." View maintenance in SQL Server is discussed in [131, 415, 416, 417, 262] (Sections 2.1, 2.4.2, and Section 5).

Teradata calls its materialized views "join indexes" (US Patent 6167399; filed 1998 and issued 2000), with a special storage organization that avoids repeating values across multiple join result rows. "Aggregate join indexes" (US Patent 6505189; filed 2000 and issued 2003) have also been introduced to support grouping and aggregation. Maintenance is automatic and incremental. Work by Luo et al. [297, 298, 299] (Sections 2.5.2 and 2.5.1) was conducted on Teradata.

None of the popular open-source database systems such as PostgreSQL, MySQL, and SQLite support materialized views at the time of this survey. However, popular recipes and extensions exist to "simulate" support of materialized views in these systems. Interestingly, in an online survey (`http://postgresql.uservoice.com/forums/21853-general`) of most wanted features in PostgreSQL, materialized views are the number one by a large margin (as of January 2012).

3

Using Materialized Views

Having discussed how to maintain materialized views, we now turn to how to use materialized views. This section begins with some background and theory on answering queries using views (Section 3.1), and then moves on to two application contexts of this problem: database query optimization (Section 3.2) and information integration (Section 3.3). There is already an in-depth survey in 2001 on answering queries using views by Halevy [203]. This section summarizes some of the points in [203], and references work published since 2001.

3.1 Background and Theory

As stated in [203], the problem of answering queries using views, in the relational setting, is as follows. Suppose we are given a query Q over a database schema $\mathcal{S}$, and a set $\mathcal{V}$ of view definitions $V_1, \ldots, V_n$ over the same schema. We are interested in the following three questions:

(1) Is it possible to compute the exact answer to the query Q using *only* the contents of the views in $\mathcal{V}$?
(2) Alternatively, what is the maximal set of tuples in the answer to Q that we can obtain from the views?

(3) If we can access both the views and the database relations, what is the best execution plan for answering Q?

Questions (1) and (2) arise when we are interested in answering queries using views for data integration (further discussed in Section 3.3). Question (3) and possibly (1) are asked when we are interested in query optimization using materialized views (further discussed in Section 3.2). This section outlines the formal foundation for answering these questions (Section 3.1.1) and presents the formal approaches and pointers to relevant literature (Section 3.1.2).

For Question (2), instead of maximizing answer "completeness" — as measured by the subset of (correct) answer tuples we can obtain — we could use a more general notion of answer "accuracy" as our objective. For example, it may be acceptable to return an approximate total sales figure that is "close enough" to the exact one. The resulting problem is studied by the approximate query processing literature, which we will discuss separately in Section 5. This section assumes the completeness-based formulation for Question (2).

Before delving into the formalism, we first illustrate the questions above with some examples.

Example 3.1 Recall from Example 1.1 the view `TotalByItemStore` and the queries `(Q1)` and `(Q1v)`. To compute the total revenue generated by each store for each item category, a business analyst may pose `(Q1)` on the base relations of the database. It can be rewritten as an equivalent query `(Q1v)` that uses the view `TotalByItemStore` instead. We reproduce the two queries here for convenience:

```
SELECT storeID, category, SUM(qty*price)           -- (Q1)
FROM pos, items
WHERE pos.itemID = items.itemID
GROUP BY storeID, category;

SELECT storeID, category, SUM(total)              -- (Q1v)
FROM TotalByItemStore, items
WHERE TotalByItemStore.itemID = items.itemID
GROUP BY storeID, category;
```

As another example, suppose the analyst is also interested in monitoring the total sales for each region. The query can be posed over the base relations as (Q2), and rewritten as (Q2v) to use the view TotalByItemStore:

```
SELECT region, SUM(qty*price)                          -- (Q2)
FROM pos, stores
WHERE pos.storeID = stores.storeID
GROUP BY region;

SELECT region, SUM(total)                             -- (Q2v)
FROM TotalByItemStore, stores
WHERE TotalByItemStore.storeID = stores.storeID
GROUP BY region;
```

It can be shown formally [117, 115] that for every possible database instance, (Q1) and (Q1v) return the same answer, and so do (Q2) and (Q2v).

Clearly, the answer to Question (1) posed at the beginning of this section is positive for the query (Q1) and view TotalByItemStore. On the other hand, as it turns out, we cannot compute the exact answer to (Q2) using *only* the view TotalByItemStore. Intuitively, the reason is that the definition of TotalByItemStore uses only the relation pos, while the definition of (Q2) uses combinations of tuples from relations pos and stores. Thus, the answer to Question (1) is negative for the query (Q2) and view TotalByItemStore.

Example 3.2 Continuing with the database of Example 1.1, consider the following two queries, which compute the total sales by store and by item, respectively:

```
SELECT storeID, SUM(qty*price)                         -- (Q3)
FROM pos GROUP BY storeID;

SELECT itemId, SUM(qty*price)                          -- (Q4)
FROM pos GROUP BY itemID;
```

It turns out that the view TotalByItemStore of Example 1.1 can be used to answer both queries without involving any other relations:

```
SELECT storeID, SUM(total)                          -- (Q3v)
FROM TotalByItemStore GROUP BY storeID;

SELECT storeID, SUM(total)                          -- (Q4v)
FROM TotalByItemStore GROUP BY itemID;
```

Thus, the answer to Question (1) is positive for both the query (Q3) and view TotalByItemStore and for the query (Q4) and view TotalByItemStore. Consequently, in both cases, the answer to Question (2) is that the maximal set of answer tuples we can obtain is the entire answer. To answer Question (3), we would need to choose between the best execution plan for the original query and that for the rewritten query; i.e., between (Q3) and (Q3v), and between (Q4) and (Q4v).

Next, suppose that instead of TotalByItemStore, we have a different materialized view, TotalInAM, defined as follows:

```
CREATE VIEW TotalInAM(itemID, storeID, total) AS
  SELECT itemID, storeID, SUM(qty*price) FROM pos
  WHERE storeID IN
    (SELECT storeID FROM stores WHERE region = 'Americas')
  GROUP BY itemID, storeID;
```

The only difference from TotalByItemStore is the presence of the WHERE clause above which restricts the computation to sales at stores in the Americas. Now, the answer to Question (1) becomes negative for both (Q3) and (Q4), given view TotalInAM — as long as pos may have tuples for stores outside the Americas — because the result of TotalInAM does not account for such tuples. For Question (2), the maximal subset of answer tuples of (Q3) that we can obtain from TotalInAM alone would be those corresponding to stores in the Americas; on the other hand, the only answer tuples of (Q4) that we can get directly from TotalInAM are those corresponding to items that have never been sold outside the Americas.

3.1.1 Formal Foundation

We now discuss the formal foundation of the questions posed at the beginning of this section. We will start with individual questions, and show that they have much theoretical underpinning in common.

Questions (1) and (3) In answering both these questions, we are faced with the following decision problem. Consider a query Q posed on database schema $\mathcal{S}$. Suppose a set $\mathcal{V}$ of views is also defined over $\mathcal{S}$. Given a particular query R defined in terms of views in $\mathcal{V}$ — for Question (1) R is defined using *only* $\mathcal{V}$, while for Question (3) R can use base relations as well — are Q and R "equivalent"? Here, R is called a (candidate) view-based rewriting of the query Q.

First, we need to clarify what "equivalence" means. For R to be a valid rewriting of Q, we generally want R to be such that the answers to R and to Q are identical on all possible database instances.[1] That is, for each database D with schema $\mathcal{S}$, we would like the result relation of Q on D to be the same as the result relation of R on D, provided that we have materialized in D all views involved in defining R. This concept of equivalence is called *equivalence modulo the views used in* (the definition of) R.

Naturally, instead of having to materialize views and test the agreement in R and Q's answer for every possible database instance, we would prefer to be able to apply to R and Q some syntactic test that ensures the equivalence *on all databases*. Such tests work by comparing the query Q to an "expansion," $R^\star$, of the rewriting R (that is equivalent to R). We obtain this expansion $R^\star$ by replacing, in the definition of R, each view name by the query defining the view. The goal in constructing the query $R^\star$ is to build a query definition that uses the names of only the base relations in the database schema $\mathcal{S}$, instead of views. That is, the definition of $R^\star$ (unlike the definition of R) is comparable with the definition of Q, in that both $R^\star$ and Q use only the relations in the schema $\mathcal{S}$. This construction of $R^\star$ from R permits

[1] In some contexts, such as in the presence of integrity constraints and/or access restrictions for users, it makes sense to focus on "just some databases" instead of all databases; see [138, 337].

one to compare "apples to apples" when deciding whether the queries R and Q return the same answers (modulo the views used in R) on all databases.

Example 3.3 Let us revisit Example 3.2. In that example, query `(Q3v)` is a candidate rewriting of the query `(Q3)` using the view `TotalByItemStore`. The equivalent expansion of the rewriting `(Q3v)` is the following query `(Q3v*)`:

```
SELECT storeID, SUM(total)                          -- (Q3v*)
FROM (SELECT itemID, storeID, SUM(qty*price) AS total
      FROM pos GROUP BY itemID, storeID)
GROUP BY storeID;
```

It is not difficult to see that `(Q3v*)` is equivalent to `(Q3)`. Thus we conclude that the rewriting `(Q3v)` is equivalent to the query `(Q3)` modulo the view `TotalByItemStore`.

Consider now another candidate rewriting, `(Q3v2)`, which is defined using the view `TotalInAM` of Example 3.2:

```
SELECT storeID, SUM(total) FROM TotalInAM        -- (Q3v2)
GROUP BY storeID;
```

The expansion of `(Q3v2)`, after some simplification, is as follows:

```
SELECT storeID, SUM(qty*price)                   -- (Q3v2*)
FROM pos GROUP BY storeID
HAVING storeID IN
  (SELECT storeID FROM stores WHERE region = 'Americas');
```

Since the queries `(Q3)` and `(Q3v2*)` are not equivalent, the rewriting `Q3v2` is not equivalent to the query `(Q3)` modulo the view `TotalInAM`.

As we have just discussed, determining view-based equivalence of queries to their view-based rewritings reduces to determining the equivalence of the queries to the expansions of the rewritings. For this reason, advances in finding equivalent view-based rewritings of the queries of interest can happen only when advances have been made in developing syntactic tests for equivalence of the underlying queries.

Question (2) Compared with Questions (1) and (3), Question (2) further motivated in Section 3.3, calls for a different decision problem. Instead of testing whether a view-based rewriting R is equivalent to a query Q, we ask whether R is "maximally contained" in Q. As in the case of testing equivalence, we first obtain $R^\star$, the expansion of R that is defined using only base relations. Formally, we say that $R^\star$ *is contained in* Q if, for every possible database D, all tuples in the answer to $R^\star$ also belong to the answer to Q on D. Further, we say that R *is contained in* Q *modulo the views* if R's expansion, $R^\star$, is contained in Q. Finally, given a query language $\mathcal{L}$ and a set $\mathcal{V}$ of views, we say that R *is maximally contained in* Q *with respect to the language* $\mathcal{L}$, if (1) R is expressed in the language $\mathcal{L}$ using only the views in $\mathcal{V}$, (2) R is contained in Q modulo $\mathcal{V}$, and (3) there is no query R' such that R' satisfies the above two conditions and R' contains but is not equivalent to R. (See [203, 271, 388] for more detailed formal discussion of maximally contained rewritings.)

As an illustration, consider again Example 3.3. Suppose $\mathcal{V} = \{\texttt{TotalInAm}\}$, i.e., only the view `TotalInAm` is available and we have no direct access to the base relations. In this case, `(Q3v2)` would be the maximally contained rewriting of the query `(Q3)`, with respect to language of rewritings that expresses SQL queries. Thus, `(Q3v2)` represents our best option for extracting the answer out of the given views. On the other hand, if we additionally have the view `TotalByItemStore`, i.e., $\mathcal{V} = \{\texttt{TotalByItemStore}, \texttt{TotalInAM}\}$, then the rewriting `(Q3v)` of Example 3.2 is the maximally contained rewriting of `(Q3)` with respect to the same language of rewritings.

As we can see from the above definitions, when we are given a query Q and a view-based query R, the problem of determining whether R is a maximally contained rewriting of Q (with respect to the given language of rewritings) reduces in part to the problem of whether the expansion $R^\star$ is contained in Q. Thus, in parallel with the case of (view-based) equivalence of queries and their potential rewritings discussed earlier for Questions (1) and (2), we want to rely on syntactic tests for query containment.

Discussion We have shown that the topics of query equivalence and containment are vital to the questions about how to use views. Because

of their importance to query optimization and data integration, these topics have been studied extensively by database researchers. Query equivalence is well understood for SQL select–project–join queries using the `DISTINCT` keyword and for their unions [84, 345], for queries with arithmetic comparisons [244, 245, 387], and for certain types of queries with grouping and aggregation [111, 116]. For basic references on general query containment, see [106]. Articles [134, 137] and book [1] provide excellent overviews and pointers for the problems of query equivalence and containment in the presence of integrity constraints.

Research on query equivalence and containment has traditionally assumed a set-based query evaluation semantics, where both database relations and query answers are relations without duplicate tuples. However, SQL by default uses bag semantics, where duplicates are allowed and their counts matter. An important direction of research is the determination of equivalence and containment between queries using query evaluation semantics reflecting real SQL. Please see [95, 112, 114, 223] for query equivalence and containment results in this direction. The complexity of the containment of SQL select–project–join queries *not* using the `DISTINCT` keyword and not using subqueries is an open problem as of 2011; please see [111] for an introduction to the problem and for relevant pointers.

3.1.2 Formal Approaches and Further Pointers

Now let us return to the questions formulated in the beginning of Section 3.1. Consider Question (1): is it possible to compute the exact answer to the query Q using *only* the contents of the views in $\mathcal{V}$? Suppose that we have available an algorithm, call it $\mathfrak{A}$, for determining the equivalence (modulo the views) of queries to view-based rewritings. In general, queries may be defined in a certain language, $\mathcal{L}_q$, rewritings may be defined in another query language, $\mathcal{L}_r$, and views may be defined in yet another query language, $\mathcal{L}_v$. Then one approach to rewriting queries defined in $\mathcal{L}_q$ using views would be to develop a generate-and-test algorithm, call it $\mathfrak{B}$, that, given a query Q in that language and a (generally finite) space of views in the language $\mathcal{L}_v$, would generate candidate rewritings in the language $\mathcal{L}_r$ using the given

views. For each such candidate "logical" rewriting R that the algorithm $\mathfrak{B}$ generates, we would then be able to use the algorithm $\mathfrak{A}$ as a subroutine for testing whether the expansion $R^\star$ of R is equivalent to the query Q. Then each rewriting R that passes the test is a candidate equivalent rewriting of the query Q modulo the given views.

For Question (2), a similar approach can be taken, where an algorithm $\mathfrak{A}$ would be used as a subroutine for testing *containment* (as opposed to equivalence), both of rewritings in queries, and between rewritings themselves. As observed in [203], such a generate-and-test approach is more applicable in data integration than in query optimization, as logical rewritings do not have "built-in" cost estimation, which is an important parameter in query optimization. We consider in Section 3.2 those rewriting-building approaches that are more suitable for query optimization and for answering Question (3).

As an example, each of the three query languages $\mathcal{L}_q$, $\mathcal{L}_r$, and $\mathcal{L}_v$ could be the language of SQL select–project–join queries using the `DISTINCT` keyword, or, to state it another way, the language of conjunctive queries under the set semantics for query evaluation. Then the classic work [84] provides the algorithm $\mathfrak{A}$ for checking whether the expansion of each rewriting R is equivalent to the given query Q. Further, the *Chase & Backchase* algorithm due to Deutsch et al. [137, 138] is a sound and complete algorithm $\mathfrak{B}$ for finding all equivalent conjunctive rewritings of a given conjunctive query using the given conjunctive views.

Articles [203, 274, 388] provide detailed overviews of algorithms for answering and rewriting queries using views, for the cases where an equivalent or contained rewriting is sought, and supply pointers. (Table 1 in [203] contains many helpful references, separately to formal results for different combinations of the languages for the queries and views.) Generally, major advances have been made over the years in formal studies of view-based query rewriting and answering. Unfortunately, an in-depth discussion of the relevant projects and publications would call for a separate full-length survey, which could (at the very least) introduce the formal problems studied on these topics. In the meantime, we recommend [202, 203] as good entry points into the subject matter, as these articles outline the relevant research results

known as of 2001 and give references to the papers in or before that year. What we do in this gateway survey is to complement the references given in [202, 203], by making a best effort attempt at providing a list of some representative pointers to more recent work related to the theory of query rewriting and answering using views. We do so below by citing and categorizing papers that appeared (mostly) in or after the year 2001. Note that some papers belong to more than one category.

- Problems concerning query containment in the presence of results of views or of view definitions [44, 73, 414], including the problem of answering questions using authorization views in security applications [337, 414];
- The problem of answering queries using views in the presence of grouping and aggregation in the definitions of the queries or views [115, 118, 182, 183, 219];
- Problems concerning answering queries using views where query or view definitions involve various other query language constructs [11, 13, 70, 80, 136, 150, 394], or otherwise revisiting the problem of answering queries using views [121];
- Problems concerning answering queries using views in the presence of integrity constraints [80, 136, 156, 219];
- Problems concerning optimizing various metrics for query rewritings in terms of views [110, 304];
- Problems related to the role of the "information content" of views in query answering [72, 74, 75, 76, 303, 310, 354];
- Problems concerning desirable transformations of a set of views [165, 264, 265, 275].

3.2 Incorporation into Query Optimization

Conceptually, for a given query posed on a data-intensive system, the task of the query optimizer in the system is to generate reformulations of the query, and to choose the reformulation whose cheapest (for this reformulation) execution plan is the cheapest among all the reformulations generated. In this process, the costs of execution plans are *estimated* by the optimizer, rather than computed exactly. For

general introduction to query optimization and reformulation, please see [86, 214, 224, 229] and references therein.

The idea of using materialized views in query optimization comes from the simple observation that if a view V is helpful in evaluating a subexpression of a given query, then the materialized result of V already contains, in the *precomputed* form, some contribution to the answer to this query subexpression. Thus, by reformulating the query using V, we can avoid computing some query subexpression, or at least some part of it, by using the materialized result of V.

For instance, Example 1.2 shows how reformulating the query `(Q1)` into `(Q1v)` using the materialized view `TotalByItemStore` leads to a dramatically cheaper execution plan. The argument is based on simple cardinality estimates — `TotalByItemStore` is orders of magnitude smaller than the base table `pos` and therefore cheaper to aggregate over. The same argument applies to reformulating `(Q2)` into `(Q2v)` (Example 3.1) and to reformulating `(Q3)`, `(Q4)` into `(Q3v)`, `(Q4v)` (Example 3.2) using the same materialized view.

This idea brings us to the notion of a view "being helpful in evaluating a query subexpression." In the examples above, we can show formally [115, 117] that there exist correspondences between the definition of the view `TotalByItemStore` and the definitions of the queries `(Q1)`–`(Q4)`, which permit us to use the view in reformulating and then evaluating these queries. In general, such correspondences ("view matchings") are syntactic criteria for the usability of a given view in rewriting a given query. For a good introduction to how to determine whether a view is usable in rewriting a query, see Section 4 of [203].

Now if our goal is to use views in evaluating queries more efficiently, then the query optimizer at hand needs to be extended with view matchings in some form. The extension of the optimizer needs to be such that all those execution plans for a query that uses views and that are considered by the optimizer must represent *equivalent* (rather than just contained, as in data integration) rewritings of the query. On the other hand, unlike the rewritings that are used in data integration, rewritings considered in query optimization do not have to be in terms of *only* the views. (As an illustration, the efficient rewriting `(Q1v)` of query `(Q1)` in Example 1.1 is defined using both a view,

`TotalByItemStore`, and a base relation, `items`.) Finally, in determining whether to evaluate a given query using a given view (or views), the optimizer must make decisions based on the cost of the execution plans for the rewritings. It is possible for a rewriting involving views to have only expensive execution plans; in that case, the optimizer must not choose such a rewriting in the presence of cheaper execution plans for other reformulations of the query.

In [203], Halevy provided a good overview of the work [43, 90, 167, 185, 363, 383, 384, 407] on incorporating views in commercial query optimizers, both of the System-R type and of other types (such as transformational). The exposition is accompanied by detailed examples. For recent work on view-inclusive query optimization with commercial prototypes, please see [131, 416, 261, 263]. The related work section of [131] provides an overview of, and pointers to, significant work in the area of determining when a materialized view is usable in query answering, back to the work [260] published in 1985. Other recent research results on rewriting queries using views in optimization include [7, 12, 168, 323, 324]. Please also see [188, 267, 285] for other introductions to and extended references on the topic, including query optimization in the online analytical processing (OLAP) setting.

3.3 Using Views for Data Integration

Data integration [45, 139, 270] is one of the most prominent applications of views. In data integration applications, users may have no control over a certain assortment of data sources, but would like to pose queries on the information that is stored collectively in all these data sources. In such a setting, it is common for a "data hub" to provide a *global schema* that serves as a uniform interface to all data sources, without sacrificing their autonomy. To this end, correspondences, or "schema mappings," must be established between the global schema and the local schema of each of the data sources. A *wrapper* at each source is used for querying and/or extracting data from this source and transforming them for integration. The data hub can either be a *mediator*, which integrates data on demand by translating user queries against the global schema into queries against sources and returning the integrated

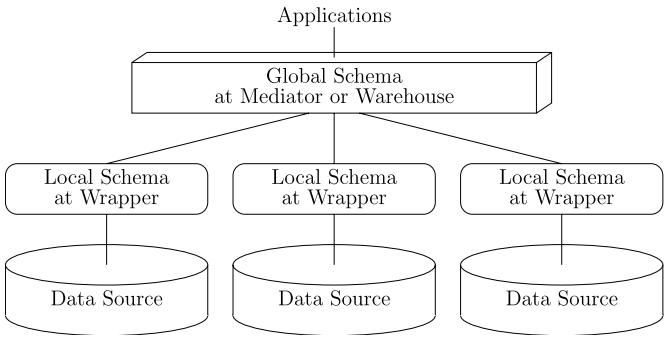


Fig. 3.1 Architecture of a data integration system.

results, or a *warehouse*, which eagerly extracts data (and changes) from data sources, and stores (and maintains) the integrated data in a central repository for querying. This architecture is illustrated in Figure 3.1.

With the warehousing-style data integrating, data stored by the warehouse are essentially materialized views over source relations. Section 2 has discussed warehousing in the context of view maintenance, and Section 4 will discuss it in the context of selecting views to materialize. The remainder of this section focuses on mediator-based (i.e., integrate-on-demand) data integration.

One approach to establishing the schema correspondences is called *local-as-view* (*LAV*). With this approach, each data source is modeled as a view defined on the global schema. Then, the process of evaluating a user query over the global schema starts with the system rewriting the query in terms of the views that define the data sources. The user is interested in obtaining the exact answer to the query, but if it is not possible due to the contents available at the sources, then the user is interested in obtaining as large a portion as possible of the exact answer. To reformulate the latter desideratum, as the next-best substitute for an exact answer to his query Q, the user would like to obtain the answer to a view-based rewriting R of Q that is maximally contained in Q (i.e., Question (2) posed in Section 3.1). We emphasize that unlike in the query optimization setting (Section 3.2), here (1) the base relations in the global schema do not actually exist, so they cannot be used in rewritings; and (2) it is acceptable to return a partial answer

to the query when the exact answer is unattainable. In some settings, the integration system includes numerous data sources, so an important consideration in query rewriting is the scalability of the rewriting algorithms to a large number of views. Please see Section 6 of [203], as well as [139] and entries [201, 239, 274, 388] in [291], for more detailed overviews of specific algorithms for query rewriting in the data integration setting.

In addition to the local-as-view (LAV) approach to building schema mappings for data integration, the research literature describes the *global-as-view* (*GAV*) and the *global-and-local-as-view* (*GLAV*) approaches. While in LAV, each local schema (i.e., the schema of each data source) is described using views defined over the global schema, in the GAV approach the global schema is described using views over the local schemas. As a result, the process of evaluating a user query under GAV involves "unfolding" the query using the definitions of the global schema in terms of the source relations. This unfolding process is the same in spirit as the process of obtaining an expansion of a rewriting, which we discussed in Section 3.1. Further, the GLAV approach generalizes properly each of LAV and GLAV. Please see [239] for a good introduction to the three approaches to schema mapping, including examples and discussions of the query-processing flavors and of the issues in each of LAV, GAV, and GLAV.

More generally, for in-depth surveys of the formal aspects of using views in information integration, please see [271, 385]. Genesereth in [162] provided an extended treatment of one of the traditional approaches to view-based data integration. Work published since 2001 on the formal foundations of data integration includes [66, 67, 68, 69, 78, 135, 164, 397]. Barcelo in [41] provided an excellent survey and references for the formal problem of "data exchange," which is closely related to data integration and has been explored actively in the recent years. See also the recent book [30] for an in-depth introduction to relational and XML data exchange.

Besides the articles [203, 239] that we have used in this section, excellent overviews and bibliographies of information integration are contained in [139, 140, 188, 199, 200, 201, 204, 207, 312]. In addition, Section 21 of [158] contains an extensive introduction to information

integration, with numerous examples. Some recent specific projects related to information integration are *Piazza* [205, 206, 226, 371], *ORCHESTRA* [177, 178, 225, 227], a project centering on peer data exchange [154, 155], and *Youtopia* [198, 248].

4

Selecting Views to Materialize

Consider a data-intensive system that uses materialized views. Such a system may be a standard database system, or a distributed system integrating data from multiple sources, such as a data warehouse. For all such systems, our focus in this section is on the selection of views to materialize — which we also refer to as "view selection" — in the context of some performance objective, e.g., speeding up query processing. Examples 3.1 and 3.2 in Section 3 illustrate how views can be used in query answering. Before a data-intensive system can use materialized views, however, there must be a process in place for adding materialized views to the collection of stored database tables. Specifically, decisions must be taken on *which* views should be materialized.

One ad hoc approach to view selection would be for a database administrator to assume (or guess) that certain materialized views would be useful in the context of some performance objective, and then to define some beneficial views manually and materialize them in the system. (For instance, for the queries `(Q1)` and `(Q2)` and for the database statistics of Example 1.1, one could guess that the view `TotalByItemStore` could be beneficial in improving the performance of processing both queries.) This approach is workable if sufficient

resources are available in the system to host and to maintain all the selected materialized views. But even in this case, it is not guaranteed that the views that have been selected and materialized truly optimize the chosen performance objective.

In practice, one cannot afford to materialize all possible views in a real-life system. The reason is that (at least) two major types of system resources may be insufficient for servicing the selected views: (1) the storage space needed for the materialized views; and (2) the system costs of maintaining the materialized views (to keep them up to date with respect to changing base data, as discussed in Section 2). As a result, the problem of selecting views to materialize must be considered under *constraints*. That is, to formulate the view-selection problem for a specific use in a specific data-intensive system, one needs to choose:

- one or more performance measures that one wants to optimize; and
- one or more types of system resources whose usage (with respect to the views to be materialized) one wants to limit.

Clearly, the approach of manual view selection by database administrators is not scalable and does not guarantee that the resulting materialized views would actually improve any performance metric in a given system. We now proceed to overview, in Section 4.1, the state of the art in *automated* view selection, with the primary objective of speeding up queries. We then survey the implementation of view selection in commercial systems in Section 4.2. Note that given a set of views to materialize, we may also want to select additional views with the objective of speeding up view maintenance or reducing reliance on base data availability; we have already discussed this related problem separately in Section 2.3.2.

View selection and *index selection* are closely interrelated problems, though an in-depth discussion of index selection is out of the scope of this survey. Instead, we provide here some important pointers. Selection of indexes to create in a data-intensive system has attracted research interest for a long time; please see [88, 91, 92, 149, 169, 321, 343] and references therein. The book by Shasha et al. [357] provides an excellent overview of what performance objectives to pursue in index selection;

for a brief summary, please also see [53]. It has long been observed (e.g., [23]) that indexes can be viewed as a type of materialized views. As such, it makes sense to select views and indexes together, which has been done in a number of research projects; see [23, 194] for influential results. Indexes and views are routinely selected together in commercial schema-design tools; see Section 4.2.

4.1 View Selection to Speed Up Queries

Use of materialized views can significantly speed up queries, as illustrated in Example 1.2. When the performance measure of choice is query-processing cost, the most general scope of view selection would consider for materialization all views that may benefit *any* possible future query. Adopting this take on the view-selection problem results in formulations of the problem that, in general, cannot be solved in practice, simply because too many candidates for materialization must be considered. Thus, in specifying the view-selection problem with query-processing costs as the performance measure, it is typical to also restrict the set of queries that are to be executed more efficiently in the presence of the views that we plan to materialize. That is, when we conduct view selection with the goal of speeding up queries, we generally have to instantiate the following dimensions in the problem statement:

- we instantiate the performance measure that one wants to optimize as *query-processing costs* for a given *query workload* (generally a finite set of queries of interest); and
- we instantiate the system resources to be constrained as, for instance, *storage space*, or *view-maintenance costs* for a given *update workload*.

Note that sometimes objectives and constraints can be reversed or mixed. For example, Gupta in [192] and other authors consider an optimization objective that is a linear combination of query-processing and view-maintenance costs; see Section 4.1.1 for more pointers to their work.

In the remainder of this section, we consider two scenarios where view selection is used for speeding up queries. Section 4.1.1 discusses

the OLAP and data-warehousing scenario, where queries and views involve a lot of grouping and aggregation. Section 4.1.2 discusses more general query workloads that arise in database systems. In both areas, we consider materialized views that can be used to rewrite the given queries *equivalently* (recall the definition and discussion of equivalent rewritings from Section 3.1).

4.1.1 View Selection for OLAP and Data Warehousing

Because most OLAP queries involve grouping and aggregation over large tables, use of materialized views can result in tremendous improvements in their processing performance [89]. As illustrated in Examples 1.2, 3.1, and 3.2, dramatic performance improvements are possible when a useful materialized view, such as `TotalByItemStore`, has already carried out much of the grouping and aggregation computation needed to answer the given queries. In the remainder of this subsection, we outline the intuition behind view selection for group-by aggregation queries by progressing through the following three scenarios:

- the case of a "single base table," where all queries of interest are defined using the same single base table, without joins;
- the case of a "single join pattern," where, intuitively, all queries of interest are defined using the same "join pattern" on the same set of base tables; and, finally; and
- the general case.

We close this subsection by reviewing specific past approaches in the literature to view selection.

The Case of a Single Base Table We consider first a special case where all queries in the workload of interest are defined using the same single base table, and without using self-joins. Recall Example 3.2 of Section 3. If {`(Q3)`,`(Q4)`} is the set of all queries of interest, then two other views (besides `TotalByItemStore`) that can be considered for materialization are defined by the queries `(Q3)` and `(Q4)` themselves. If we materialize the view defined by `(Q3)`, for example, the cost of processing the query `(Q3)` would be just the cost of returning the contents

of the materialized view (which would make the cost of processing (Q3) globally minimal). Together, the three views form the search space of candidate views to materialize in this case.

We now generalize over this example. Let an instance of the view-selection problem for OLAP have a set $\mathcal{Q}$ of queries of interest. Suppose that all queries in $\mathcal{Q}$ are formulated in terms of a single base table T by grouping by some subset of the "dimension" attributes in T and aggregating the "measure" attributes in T. (Generally, additional restrictions also need to be imposed on the aggregation functions; see [113, 115, 116].) Then it is customary to use a "view lattice" [173, 212] to explicitly define the search space of all views that are candidates for materialization. Each view in the lattice is defined (1) using a `GROUP BY` clause with a subset of the dimension attributes of the table T; and also (2) using a `SELECT` clause that has all the view's `GROUP BY` attributes, as well as the aggregations that are appropriate for the queries in $\mathcal{Q}$. An example of view lattice is shown in Figure 4.1. Note that our restrictions on the queries in $\mathcal{Q}$ ensure that each of them can be answered without joins by some view from the lattice (there may be multiple such views). As an example, (Q3) can be computed from one of the views `TotalByStore`, `TotalByItemStore`, and `TotalByItemStoreDate` in Figure 4.1.

Given an instance of the OLAP view-selection problem with the set $\mathcal{Q}$ of queries of interest, and having constructed a view lattice $\mathcal{S}$ for $\mathcal{Q}$, we now define what we mean by a solution and a globally optimal

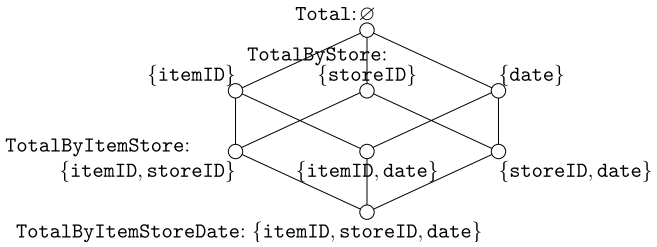


Fig. 4.1 The candidate view lattice for `pos`, assuming that: (1) the dimension attributes are `itemID`, `storeID`, and `date`; and (2) the only aggregated measure of interest is `SUM(qty*price)`. The lattice is induced by the containment relation among the subsets of dimension attributes used in `GROUP BY` (shown next to the lattice nodes).

solution to the problem. A subset S of the views in $\mathcal{S}$ is called a *candidate viewset* if S satisfies the input constraint(s) on the system resources. Then, a candidate viewset S is a *solution* to this instance of the view-selection problem whenever S (that is, all the views in S collectively) provides a lower cost of answering the queries in $\mathcal{Q}$ than the cost of answering them using just the original base tables. A solution S is *globally optimal* whenever S provides the lowest cost of answering the queries in $\mathcal{Q}$ among all the candidate viewsets.

When the view lattice is taken as the complete search space of views that are candidates for materialization, approaches to view selection differ in how they explore the search space. Note that in practical instances of the view-selection problem for OLAP, exhaustive exploration of the search space of views would result in prohibitive running times for view selection. Thus, in general, view-selection algorithms look for "good" solutions, rather than globally optimal solutions. Please see [194, 195, 212, 237] for details.

The Case of a Single Join Pattern We now take another generalization step over the case of a single base table. Suppose that in the given instance of the OLAP view-selection problem, each query of interest is defined using exactly the same set of base tables and by following the same join pattern on these tables. An instance of this case that is very frequent in practice is the scenario where the data warehouse has a "star schema" [242, 328]; that is, the schema has a single "fact table" and a number of "dimension tables," which are all related to the fact table via foreign-key references from the fact table's dimension attributes. For example, in Example 1.1, `pos` is the fact table, while `stores` and `items` are dimension tables. With a star schema, we encounter queries that are defined over the natural join between the fact table and all dimension tables along with the foreign-key references.

It is straightforward to generalize the search based on the view lattice, as explained above for the case of a single base table, to view selection for this case. Intuitively, the approach is to build a view lattice where each lattice view is defined using the same join pattern as in all queries of interest. Thus, any algorithm that works for the view-lattice setup as discussed above is also applicable in this case.

The General Case Now consider what happens when the queries of interest in a given instance of the OLAP view-selection problem do not all fit the same join pattern. As an illustration, recall the queries `(Q1)` and `(Q2)` and their rewritings using the view `TotalByItemStore`, as discussed in Example 3.1. Such rewritings are obtained by *joining* materialized views with either base tables or other materialized views. Such rewritings can be constructed automatically as described in [5, 6, 115, 117]. The authors of [5, 6] described an approach to view selection where such *join-based* rewritings are considered. In general, approaches to view selection where join-based rewritings are allowed are much more complex than view-selection methods based on the view lattice. Because of the possibilities of using a materialized view by joining with other tables, the overall number of candidate materialized views can be orders of magnitude more than in approaches based on the view lattice. Section 4.1.2 also provides the intuition behind this explosion of search space.

Approaches in Literature We now review specific approaches in the literature to view selection in OLAP and data warehousing. As we have already mentioned, Harinarayan et al. introduced in [212] the view lattice, and proposed algorithms for view selection under the storage limit using the view lattice. The follow-up work [194] considered selection of indexes alongside views, again using the view lattice. In [192], Gupta considered view selection that targets minimizing both query-processing and view-maintenance costs. In [195], Gupta and Mumick proposed methods for view selection under the constraint of view-maintenance cost; for alternative approaches to the same problem, see [279] for two heuristic algorithms and [266] for a randomized algorithm. More detailed and extended expositions of the results of [192, 194, 195] can be found in [193, 196]. Baralis et al. [40] presented algorithms aimed at reducing the size of the solution space in the view lattice; see also [322] for related results. In a related development, Karloff and Mihail in [237] reported on their complexity analysis of the view-selection problem on view lattices. Among other work, Shukla et al. proposed in [358] improvements over the algorithms of [212], and later in [359] considered view selection for a warehouse containing

multiple data cubes [173]. Later, Hanusse et al. [211] further improved the result of [358].

A separate line of work considers doing view and index selection on the view lattice, where globally optimal solutions can be found up to realistically large sizes of view lattices. Please see [32, 33, 276] for reports on this line of work by Asgharzadeh Talebi et al.

While the view lattice is appropriate for the cases of a single base table or single join pattern, it is no longer sufficient for capturing the search space of candidate views in the general case. In [192], which we have already mentioned above, a more general framework based on *AND-OR view graph* was proposed, which served as a basis for, e.g., [192, 195]. Chirkova and Li [109] studied how to select a viewset with minimal storage cost to support a given set of conjunctive queries without accessing base tables.

View selection with a focus on minimizing both query-processing and view-maintenance costs has been considered in a number of projects. In addition to the work [192] that we have already mentioned, this problem has been considered in [403, 409] and in a suite of related projects by Theodoratos et al., see [373, 374, 375, 376, 377, 378]. Excellent overviews can be found in [360, 379].

In data warehouses, views and indexes can also be selected for materialization in stages over time, to accommodate changes in the prevalent set of queries of interest. Please see work on *DynaMat* by Kotidis and Roussopoulos [249, 250], as well as [215, 326, 408]. Another issue that has been considered in conjunction with warehouse view selection is the placement of views in a distributed setting [231].

4.1.2 View Selection for General Database Systems

As a prerequisite to the view-selection problem, we need to know whether and how a query of interest can be rewritten in term of materialized views (Section 3.1). However, for general database queries and views, this question is far from obvious, which makes the view-selection problem in general very difficult in practice. To illustrate, consider the following example.

Example 4.1 Recall the database in Example 1.1 with tables `pos`, `stores`, and `items`. Suppose that there are two extra tables: `distributors(distID,city,region)` stores address information about distributors that supply items to stores, and `supply (storeID,distID,contractType)` associates each store with the distributors that supply items to the store (the relationship is many-to-many). Key attributes are underlined.

Consider a query `(Q5)` that asks for the total sales generated by "locally supplied" stores in each region, where a store is "locally supplied" if it has at least one local distributor (i.e., in the same city and region as the store):

```
SELECT region, SUM(qty*price)                          -- (Q5)
FROM pos, stores S
WHERE pos.storeID = S.storeID AND EXISTS
  (SELECT * FROM supply P, distributors D
   WHERE P.storeID = S.storeID AND P.distID = D.distID
   AND D.city = S.city AND D.region = S.region)
GROUP BY region;
```

Suppose that relatively few stores have local distributors. It would be nice to materialize the following view `StoresWithLocalDist`, which returns all store information for locally supplied stores:

```
CREATE VIEW StoresWithLocalDist(storeID,city,region) AS
  SELECT DISTINCT S.storeID, S.city, S.region
  FROM stores S, supply P, distributors D
  WHERE P.storeID = S.storeID AND P.distID = D.distID
  AND D.city = S.city AND D.region = S.region;
```

This view would enable the following equivalent rewriting `(Q5v)` of `(Q5)`, which can be cheaper to process than `(Q5)`:

```
SELECT region, SUM(qty*price)                         -- (Q5v)
FROM pos, StoresWithLocalDist V
WHERE pos.storeID = V.storeID
GROUP BY region;
```

It is not straightforward to show the equivalence, modulo the view `StoresWithLocalDist`, of the queries `(Q5)` and `(Q5v)`. The use of subqueries in `(Q5)` means that it is no longer sufficient to reject equivalence simply by inspecting the outermost `FROM` clauses of queries and view definitions. Handling duplicates is also tricky: `(Q5)` uses an `EXISTS` subquery to avoid duplicates that would otherwise be introduced by joining with `supply`, and `StoresWithLocalDist` uses `DISTINCT` to remove duplicates. In fact, we are not aware of formal results in the open literature that would provide general methods for ascertaining the equivalence of such queries and rewritings.

Criteria for Restricting the Search Space of Views As we have seen in Example 4.1, even if a materialized view can in theory help process a query more efficiency, the query optimizer of the system may not be able to discover the appropriate rewriting. Naturally, for a given set of queries of interest, we want to select to materialize only those views that can be used by the system, specifically by its query optimizer, to efficiently rewrite the queries of interest. This criterion — *implied by the query-rewriting capability* of the query optimizer — is the first criterion by which we restrict the search space of views considered for materialization. Following this criterion, it is common in the development of algorithms for view selection to restrict the languages for defining queries, views, and rewritings (recall $\mathcal{L}_q$, $\mathcal{L}_v$, and $\mathcal{L}_r$ defined in Section 3.1.2). For instance, as discussed in Section 3.1.2, there is a good understanding [108, 203, 272] of how to rewrite queries equivalently using views in the case where all three languages are the language of conjunctive queries; for a query optimizer that knows how to generate query rewritings only for this case, it would be reasonable to restrict the search space of views to those defined by conjunctive queries.

The second criterion for restricting the search space of views considered for materialization is *driven by the practical limit on the cost of searching the space*. Clearly, for the (automated) view-selection process to be practical, its running time cannot be too long. Even with well-understood algorithms for rewriting queries using views, this

process can be expensive if it needs to consider *all* admissible (i.e., syntactically appropriate) views and rewritings. For instance, even when we restrict each of $\mathcal{L}_q$, $\mathcal{L}_v$, and $\mathcal{L}_r$ to be the language of conjunctive queries, the search space of admissible views and rewritings is, in general, very large. The reason is that the admissible conjunctive views and rewritings can be expressed using *joins*. As an illustration, consider a query involving n base tables. Views that can potentially be used to rewrite it may be defined using in their `FROM` clause any nonempty subset of the n tables. On top of choosing which tables to include into `FROM`, we also get to choose the attributes in the `SELECT` clause of these views. Furthermore, when rewriting the query, we need to consider combinations of these views. In fact, it has been shown [107, 108] that if we consider in view selection *all* admissible views in this conjunctive setting, the number of views *in the rewriting* can be exponential in the size of the problem input. The results of [5, 6] for the complexity of view selection for queries with aggregation build on the above results of [107, 108]. In contrast, when only admissible views of a certain type are considered, then in the worst case, the number of views in the rewriting is polynomial in the size of the problem input [203, 272].

Approaches in Literature As a result of the two criteria that we have just discussed, most approaches put significant efforts into restricting the search space of candidate views for general queries. Agrawal et al. [23] proposed in 2000 an influential end-to-end framework for selection of views, together with indexes, for sets of queries of interest that could include both OLAP and OLTP (online transaction processing) queries. The architecture has an explicit module that defines the search space of useful views based on the given queries of interest. Another influential feature of the approach of [23] is the use of a "what-if" optimizer [20, 92] to determine which subsets of that search space are the most promising "configurations" for materialization. A configuration consists of existing as well as *hypothetical* physical structures, and for each configuration the "what-if" query optimizer is requested to return the best plan (and its cost) for executing each query of interest, as if the database had the entire configuration materialized. As is summarized in [20], this approach is possible because the query

optimizer does not require the presence of a fully materialized view or index to be able to generate plans that use the view or index; instead, the optimizer uses only the metadata stored in the system catalog and the relevant statistics for each hypothetical structure, often gathered via sampling.

As pointed out in [23], determining the "quality" of an output configuration (i.e., of a set of views and indexes recommended for materialization) is a hard problem. Here, quality is understood as the costs of executing the queries of interest using the output configuration, with respect to the costs using a globally optimal solution/configuration. (The definitions of "solution" and of "globally optimal solution" for the case of general queries parallel the definitions of the same terms in Section 4.1.1.) As a result, many practical approaches to view (and index) selection focus on producing "good" solutions as opposed to globally optimal solutions. Not surprisingly, it has been shown experimentally [246, 247] that it is possible to improve the solution quality of heuristic view-selection algorithms, such as the one used in [23], when systematic approaches (e.g., dynamic programming and integer linear programming in case of [246, 247]) are used.

The approach of [23] has been implemented as part of the cost-based physical database design tool in Microsoft SQL Server. The implementation has since been enhanced with a number of other methods associated with automated selection of views for consideration by the DBA to materialize. Detailed reports on these methods and their implementation and experimental evaluation are available in the open literature; see, for instance, work by Bruno and Chaudhuri in [56, 57, 58, 60, 61, 62], as well as references for the Microsoft Research *AutoAdmin* project later in Section 4.2.

Given that recent techniques for selecting physical structures have become increasingly complex, Bruno and Chaudhuri in [55] called for a radical simplification of the assumptions used in previous approaches, and took a step in this direction with the proposal of a new framework for the physical database design problem.

Echoing our discussion at the end of Section 3.1.1 concerning query equivalence and containment under set versus bag semantics, we note here that most formal approaches to the view-selection problem assume

set semantics. However, SQL by default uses bag semantics. As we saw in Example 4.1, proper handling of duplicates can be tricky under bag semantics. Besides work discussed in Section 3.1.1 on query equivalence and containment under bag semantics, view selection for conjunctive queries under bag and bag-set semantics is considered in [9]; view selection for aggregate queries, where bag and bag-set semantics arise naturally, is considered in [5, 6].

A number of other developments on view selection are worth noting. Kimura et al. [243] considered exploiting correlation in attribute values when selecting materialized views and indexes. Chaves et al. [98] studied the view-selection problem for distributed databases and developed a solution based on genetic programming; additional pointers to view selection work in the distributed setting can also be found in [98].

4.2 Implementation in Commercial Database Systems

It has long been recognized that the physical database design can have a big impact on the overall performance, both of the database system and of applications. Significant research and implementation efforts have concentrated on designing algorithms and, more generally, system architectures that would enable automatic tuning, also called *self-tuning* or *self-management*, of various aspects of the database system. Automation of physical database design is one aspect of such self-tuning. The objective of these efforts has been to make obsolete the error-prone, lengthy, and expensive manual tuning process. Please see [54, 285, 356, 357] for in-depth coverage of physical database design and of database tuning, and [63, 93, 94, 96] for introductions to automatic tuning of database systems. Architectures for self-tuning of data-intensive systems and lessons learned during the history of database tuning in the industry are discussed in [93, 97, 396].

To address the problem of self-tuning of data-intensive systems, Microsoft Research launched in 1996 its *AutoAdmin* research project [20]. This project has achieved a number of significant results in self-tuning — including those on automated view selection, which are most relevant to this section — over the years. The tuning technology developed within *AutoAdmin* has been incorporated successfully

into Microsoft SQL Server and into other DBMS products. One of the achievements of *AutoAdmin* has been the development of an end-to-end architecture for physical database design, including the "what-if" optimizer [92] in the loop during the design process. Such "what-if" analysis has been incorporated by major players in the industry; see, for instance, [126, 282]. Paper by Agrawal et al. [20] provides an excellent overview, as well as an extensive bibliography, of the research and commercial outcomes of the *AutoAdmin* project as of 2006. As summarized in [20], the research results from *AutoAdmin* on physical database design tuning were first implemented in Microsoft SQL Server 7.0 in 1998, and have been part of Microsoft SQL Server ever since. For details on specific directions in database tuning and view selection within the project over the years, please see [87] for an early overview of the self-tuning technology, [21, 22, 23, 24, 59, 91] for discussions over the years of the central ideas of the Microsoft database-tuning advisor, and [55, 58, 61] for approaches to solving various problems in physical database design. Additional pointers on view selection in *AutoAdmin* are provided in Section 4.1.2.

Work on self-management in IBM DB2 products started with the development of the Index Advisor, which was first implemented in version V6 of the DB2 Universal Database [386]. The DB2 Autonomic Computing project was subsequently created as a joint effort between the IBM Almaden Research Center and the IBM Toronto Software Lab, and later the Watson Research Center. The focus of the project has been on making existing database products easier and cheaper to manage, generally by automating or semiautomating administrative tasks. Excellent overviews of the project can be found in [282, 283]. Specific directions of the project are followed in [281, 284, 293]. In particular, view and index selection and related problems in DB2 are summarized in the following papers: [386] (index selection), [422, 423] (view and index selection), [142] (selection of statistical views), and [277] (autonomic placement of materialized views for load balancing). Some of the work on view selection in DB2 has its roots in the work [407] by Zaharioudakis et al., which described an approach to rewriting complex decision-support queries with grouping and aggregation using materialized views.

In Oracle, views can be defined and materialized either manually or using Oracle Enterprise Manager 11g. This software suite includes an SQL Tuning Advisor and an SQL Access Advisor. Together, these tools use proprietary techniques to analyze representative query workloads with an eye on performance improvements in query evaluation. The output of this analysis is in the form of recommendations, along with a rationale for each recommendation and its expected performance benefit. The recommendations include advice on which indexes and views can be materialized to improve the system performance. Publications in academic venues on automatic tuning in Oracle include [42, 125, 126].

A physical design advisor implemented in PostgreSQL is described in [160], and index tuning in PostgreSQL is introduced in [280]. Performance monitoring for tuning Ingres is discussed in [380].

Advances in self-tuning of data-intensive systems are generally followed by the *International Workshop on Self-managing Database Systems* (*SMDB*), which has been running on a yearly basis since 2006. See [2, 27, 28, 36] for reports on some of the past SMDB workshops.

5

Connections to Other Problems

We now explore the connections between materialized views to several other research problems. This section is not intended to be a survey of these problems. Therefore, the references herein are not complete; we provide only enough pointers in order to make the connection with materialized views clear. We also note that the list of connections covered in this section is by no means comprehensive. Rather, readers should regard them as examples and leads, and are encouraged to identify new connections as well as strength existing ones.

Data Stream Processing The past decade has seen growing popularity of *data stream* systems [15, 309]. Under the stream model, data arrive continuously in order, while queries run continuously and return new results as new data arrive. Because of high data volume, it is infeasible for the system to assume access to historical data for processing queries. Stream queries bear strong resemblance to incrementally maintained views. The maintenance procedure for a view can be seen as a stream query — its input streams are the sequences of base table modifications, and its output stream is the sequence of resultant modifications to the view contents. Indeed, recent work by Ghanem et al. [163]

adopted this interpretation of stream queries when adding view support to data stream systems.

The restriction that the system can store only a bounded amount of data online in a "scratch space" to process stream queries translates to a stronger notion of self-maintenance (Section 2.2.2). Here, given a view V corresponding to a stream query, we want to find a set of views $\mathcal{A}$ of bounded space (to maintain in the scratch space) such that we can incrementally maintain $\{V\} \cup \mathcal{A}$ using only base table modifications and the contents of $\mathcal{A}$ (but not V itself). Compared with the standard practice of keeping the recent window of stream data online, this definition allows other forms of auxiliary data to be considered. Extending it to run-time self-maintenance (Section 2.2.3) is also natural.

The connection does not stop with semantics, of course. For example, Jagadish et al.'s work on *chronicles* [228] has been widely cited by works on view maintenance and on stream processing. As with streams, append-only modifications with monotonically increasing sequence numbers greatly simplify and enable processing with bounded space. A survey of stream joins by Xie and Yang [398] pointed out more connections: exploiting constraints to reduce stream join state [37] is analogous to using them in self-maintenance, and various stream join techniques [35, 47, 236, 391] naturally translate to ideas in optimizing view maintenance queries. In terms of system building, recent efforts by Ghanem et al. on adding views to data stream systems [163] and by Kennedy et al. on building *dynamic data management systems* [26, 241] represent the first steps toward uniting view maintenance and stream processing.

Approximate Query Processing Traditionally, database views provide precise answers (with respect to some consistent database state), but many uses for views can tolerate some degree of inaccuracy just as others can tolerate staleness (Section 2.4.2). There has been extensive work on approximate querying. For example, much research on data streams involves approximate query answering, e.g., *load shedding* [34] and maintaining statistical summaries [16] for streams. Sampling [315] has also been studied extensively for its applications to

approximate query answering and cardinality estimation for query optimization. The myriad of approximate query processing techniques using various types of synopses [123] have thus laid a solid algorithmic foundation for studying "approximate materialized views." Such views have been considered by Larson et al. [258, 259] to support cardinality estimation for query optimization, and by Jermaine et al. [230, 232, 233, 329] as well as Gemulla and Lehner [161] to support approximate queries. It would be interesting to see whether we will be able to approach the three basic questions for approximate materialized views — how to maintain them, how to use them, and how to select them — with similar levels of generality and automation as traditional materialized views.

Scalable Continuous Query Processing Triggers, constraint and event monitors [325], and subscriptions in publish/subscribe systems [146] all in essence entail continuous queries, whose processing resembles view maintenance as discussed above in the context of data streams. A trigger or monitor can be seen as a query that continuously returns a Boolean value or a sequence of database events matching the trigger or monitor condition. In traditional publish/subscribe, subscriptions are just selections over events; in modern systems, however, subscriptions can be complex continuous queries involving join and/or aggregation of the event history.

One focus of the work in these areas has been scalability in the number of continuous queries. The key to scalability is group processing. Work on view maintenance has considered sharing common subexpressions and other multi-query optimization techniques (Section 2.1.3). However, to scale up to thousands or even millions of continuous queries, as some systems (such as publish/subscribe) must do, exploiting identical subexpressions is no longer sufficient. We need more general methods for identifying queries whose answers are affected by an event without examining all queries. A powerful observation is the interchangeable roles of queries and data, which, for example, was systematically exploited by Chandrasekaran and Franklin in building a continuous querying system [85]. Continuous queries can be treated

as data, while each new data modification can be treated as a query requesting the subset of continuous queries affected by its arrival. Thus, it is natural to apply indexing and processing techniques for data to continuous queries. For example, consider the problem of maintaining many continuous range-selection queries of the form $\sigma_{a_i \leq A \leq b_i} R$, where A is a column of table R and a_i and b_i are parameters that differ across views. These queries can be indexed as a set of intervals $\{[a_i, b_i]\}$. Given an insertion $\hat{r}$ into R, the set of affected queries are exactly those whose intervals contain $\hat{r}.A$. With an appropriate index, this operation can be done in logarithmic time without examining all queries.

The idea of indexing queries as data has been applied to trigger processing (e.g., [210]), continuous query processing (e.g., [101, 301]), and publish/subscribe (e.g., [25, 147]). The same idea can be applied to scale up the number of materialized views as well. Indeed, some early work on multiple select–project view maintenance [352, 353] (Section 2.1.3) uses simple predicate indexing. Techniques for indexing more complex queries beyond equality- or range-selection queries have also been developed (e.g., [14, 133]); it would be interesting to see how well they apply to view maintenance.

Caching As noted in Section 1, one recurring theme in computer science is the use of derived data to facilitate access to base data. Like materialized views, caches are derived data; thus, we are faced with the same questions of what to maintain, how to maintain, and how to use them. Traditionally, caching is done at the object level: a replacement policy controls which objects are cached; maintenance involves invalidating stale object copies or refreshing them with new contents; use of the cache is limited to accessing objects by identifiers. If the cache receives a declarative query, e.g., a selection involving non-id attributes, we cannot tell whether the cache provides a complete answer, and therefore must query the source data, which is expensive in a distributed setting. Dar et al. [129] proposed *semantic caching*, which combines the ideas of caching and materialized views. A semantic cache remembers the semantic descriptions of its contents as view definitions, so we can determine the completeness of query answers (e.g., using the techniques from Section 3), and query the source only when needed. This

idea has been applied to a wide range of settings, including caching for web and mobile as well as other types of distributed systems. For example, Li et al. [277] considered automatic placement of materialized views, which serve as caches, in multitiered database systems. See the survey of web caching by Labrinidis et al. [255] for additional pointers.

Conversely, traditional caching has also inspired interesting variations on the application of materialized views. For example, instead of maintaining materialized views, Candan et al. [77] proposed invalidating them whenever their contents are affected by base table modifications, like an invalid-on-write cache. For another example, while materializing a view traditionally implies fully materializing its contents, Luo [296] and Zhou et al. [417] proposed materializing only the subset of the contents most frequently queried, like caching. Partially materialized views have lower storage and maintenance costs. They may still support queries requesting specific subsets of rows, and can also be useful when complete query results are not needed (or not needed immediately). In [296], the multidimensional space of view contents is partitioned into regions called "basic condition parts" and selectively materialized (an upper limit is further placed on the number of rows materialized for each region). Answering a query using these regions is efficient because they are defined by simple multicolumn selections. In [417], "control tables" dictate what contents to cache; the materialized part of a view is defined essentially by the semijoin of the view with a control table. The system automatically checks if a query posed against a view can be answered by its materialized part. For deciding what is materialized, both works used policies that resemble cache replacement policies.

Indexes Indexes are another form of derived data used heavily in database systems. A single-table index is much like a project view, while a join index is like a project–join view. One traditional distinction between indexes and materialized views is that indexes consider how to store their contents physically using efficient data structures. However, as we have seen in Section 2.3.1, a lot of work on materializing views also considers efficient physical representation; therefore, this distinction from indexes is rather blurred. Because of the similarity between

materialized views and index, they share many ideas and techniques. From the aspect of maintenance, the choice of incremental view maintenance versus recomputation and that of immediate versus deferred view maintenance also exist for indexes, especially variant indexes designed for complex analytical query workloads, e.g., [318]. As discussed in Section 4, the problem of view selection is closely related to that of index selection; in fact, they are routinely dealt with together by commercial database tuning tools. As another example, earlier in this section, we have seen how partially materialized views incorporate the principles of caching in deciding what to materialize; a similar idea has long been applied to indexes [355, 365].

Provenance Study of *provenance* (or *lineage*) [105, 368] is concerned with understanding the derivation of some data item as the result of a sequence of database operations. Sarma et al. [348] pointed out that many incremental view maintenance algorithms — particularly those handle deletions, e.g., derivation counting in Section 2.1.1 — exploit some technique or auxiliary information that is essentially lineage. Conversely, incremental view maintenance techniques (Section 2.1) have been applied to provenance maintenance [176], and materializing auxiliary views for view maintenance (Section 2.3.2) can help make lineage tracing more efficient [124].

Other Connections It is natural to extend materialized views to other data models; there has been a lot of work on views for spatiotemporal data, semistructured data (such as XML), and probabilistic data, which is beyond the scope of this survey. Here, we briefly note a couple of lines of recent work that push the envelope of incrementally maintained materialized views — or more generally, derived data with declarative specifications — to novel application contexts. To simplify development and enable optimization of AJAX-based web applications, Fu et al. [152, 153] proposed treating data-driven AJAX web pages as views, which could then be specified declaratively and maintained automatically and incrementally. Loo et al.'s work on *declarative networking* [294] allows declarative programming of network protocols and services, enabling automatic optimization and safety checks. One

essential feature of declarative networking is the incremental execution of declarative programs as the underlying network changes. As the popularity of declarative programming grows, we expect to see more potential applications of the ideas from materialized views.

6

Conclusion and Future Directions

In this monograph, we have provided an introduction and reference to the topic of materialized views, which are essentially queries whose results are materialized and maintained to facilitate access to base tables. They are a form of derived data that come with semantic descriptions of their contents — view definition queries — which enable declarative querying and various maintenance optimizations. We have covered three fundamental problems that arise in the study of materialized views — or any form of derived data — how to maintain them, how to use them, and how to decide what to maintain. We have pointed out their connections to a few other research problems, and demonstrated the benefit of cross-pollination of ideas.

At the time of this writing, materialized views are considered a mature technology in the traditional SQL setting, with fairly sophisticated implementations by most commercial database systems. Where, then, does the research on materialized views go from here? Looking at the recent work on or related to materialized views, we believe that there are three promising directions.

First, the idea of materialized views naturally extends to different data models and query languages, from the more traditional

object-oriented and spatiotemporal ones to the more recent semistructured and probabilistic ones. While we have limited the scope of our survey mostly to SQL views, there exists a vast body of literature on materialize views in settings old and new alike. As two examples of work in this direction, materialized views have been considered in semistructured (e.g., [8, 10, 31, 38, 52, 71, 79, 81, 82, 99, 103, 143, 148, 166, 171, 190, 191, 216, 218, 257, 292, 302, 319, 344, 349, 350, 366, 369, 400, 401, 410]) and probabilistic database settings (e.g., [127, 128, 336]).

One particularly interesting problem along this direction is how to extend the idea of materialized views to large-scale data-parallel processing which has become prevalent in recent years, thanks to the popularity of cloud computing and parallel programming frameworks such as *MapReduce* [130]. We are just beginning to see system support for incremental computation in this setting (e.g., [46, 122, 316, 330]). Although *MapReduce* itself is not declarative in the sense of query languages, more declarative interfaces have been built on top of it (e.g., [317, 381]), thereby opening up opportunities for automated approaches toward optimizing incremental computation and deciding what to compute incrementally in order to support a given workload.

Second, we are seeing increasing cross-pollination between the research on materialized views and other topics. A natural step following cross-pollination would be consolidation of ideas and methods for various forms of derived data under various settings into a unified framework. With such a framework, we can promote sharing of data and processing among traditionally separate tasks, and make better joint decisions on the materialization and maintenance of derived data in order to achieve overall performance objectives under resource constraints. One example in this direction is the development of automated database tuning tools that consider both indexes and materialized views, which has been well underway (Section 4). However, much more remains to be done. Specifically, it would be nice for the framework to also consider, holistically, many issues discussed in Section 5: scalability to a large number of views and triggers, management of synopsis in support of both query optimization and approximate query answering, unifying materialized views and caching, etc. The additional complexity introduced by generalization needs to be better understood

and tamed, but the potential benefit of a holistic approach makes this challenge worthwhile pursuing.

Finally, driven by the widening gap between the cost of compute power and that of manpower, declarative and data-driven programming is steadily gaining popularity — concrete examples include declarative web development and declarative networking discussed at the end of Section 5. We expect this trend to give rise to more potential applications of ideas rooted in materialized views — because among various forms of derived data, materialized views are special in having declarative, semantic descriptions of their contents.

References

[1] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995.

[2] A. Aboulnaga and K. Salem, "Report: 4th international workshop on self-managing database systems (SMDB 2009)," *IEEE Data Engineering Bulletin*, vol. 32, no. 4, pp. 2–5, 2009.

[3] B. Adelberg, H. Garcia-Molina, and B. Kao, "Applying update streams in a soft real-time database system," in *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pp. 245–256, San Jose, California, USA, May 1995.

[4] M. E. Adiba and B. G. Lindsay, "Database snapshots," in *Proceedings of the 1980 International Conference on Very Large Data Bases*, pp. 86–91, Montreal, Quebec, Canada, October 1980.

[5] F. Afrati and R. Chirkova, "Selecting and using views to compute aggregate queries," in *Proceedings of the 2005 International Conference on Database Theory*, pp. 383–397, Edinburgh, UK, January 2005.

[6] F. Afrati and R. Chirkova, "Selecting and using views to compute aggregate queries," *Journal of Computer and System Sciences*, vol. 77, no. 6, pp. 1079–1107, 2011.

[7] F. Afrati, C. Li, and J. Ullman, "Generating efficient plans for queries using views," in *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pp. 319–330, Santa Barbara, California, USA, June 2001.

[8] F. N. Afrati, R. Chirkova, M. Gergatsoulis, B. Kimelfeld, V. Pavlaki, and Y. Sagiv, "On rewriting XPath queries using views," in *Proceedings of the 2009*

International Conference on Extending Database Technology, pp. 168–179, Saint Petersburg, Russia, March 2009.

[9] F. N. Afrati, R. Chirkova, M. Gergatsoulis, and V. Pavlaki, "View selection for *real* conjunctive queries," *Acta Informatica*, vol. 44, no. 5, pp. 289–321, 2007.

[10] F. N. Afrati, M. Damigos, and M. Gergatsoulis, "Union rewritings for XPath fragments," in *Proceedings of the 2011 International Database Engineering and Applications Symposium*, pp. 43–51, Lisbon, Portugal, September 2011.

[11] F. N. Afrati, C. Li, and P. Mitra, "Answering queries using views with arithmetic comparisons," in *Proceedings of the 2002 ACM Symposium on Principles of Database Systems*, pp. 209–220, Madison, Wisconsin, USA, June 2002.

[12] F. N. Afrati, C. Li, and J. D. Ullman, "Using views to generate efficient evaluation plans for queries," *Journal of Computer and System Sciences*, vol. 73, no. 5, pp. 703–724, 2007.

[13] F. N. Afrati and V. Pavlaki, "Rewriting queries using views with negation," *AI Communications*, vol. 19, no. 3, pp. 229–237, 2006.

[14] P. K. Agarwal, J. Xie, J. Yang, and H. Yu, "Scalable continuous query processing by tracking hotspots," in *Proceedings of the 2006 International Conference on Very Large Data Bases*, pp. 31–42, Seoul, Korea, September 2006.

[15] C. C. Aggarwal, ed., *Data Streams: Models and Algorithms*. Springer, 1st ed., Novermber 2006.

[16] C. C. Aggarwal and P. S. Yu, "A survey of synopsis construction in data streams," in Aggarwal [15], pp. 169–207.

[17] D. Agrawal, A. E. Abbadi, A. Mostéfaoui, M. Raynal, and M. Roy, "The lord of the rings: Efficient maintenance of views at data warehouses," in *Proceedings of the 2002 International Symposium on Distributed Computing*, pp. 33–47, Toulouse, France, October 2002.

[18] D. Agrawal, A. E. Abbadi, A. K. Singh, and T. Yurek, "Efficient view maintenance at data warehouses," in *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pp. 417–427, Tucson, Arizona, USA, May 1997.

[19] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan, "Asynchronous view maintenance for VLSD databases," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pp. 179–192, Providence, Rhode Island, USA, June 2009.

[20] S. Agrawal, N. Bruno, S. Chaudhuri, and V. R. Narasayya, "AutoAdmin: Self-tuning database systems technology," *IEEE Data Engineering Bulletin*, vol. 29, no. 3, pp. 7–15, 2006.

[21] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala, "Database tuning advisor for Microsoft SQL Server 2005," in *Proceedings of the 2004 International Conference on Very Large Data Bases*, pp. 1110–1121, Toronto, Canada, August 2004.

[22] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala, "Database tuning advisor for Microsoft SQL Server 2005: demo," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pp. 930–932, Baltimore, Maryland, USA, June 2005.

[23] S. Agrawal, S. Chaudhuri, and V. R. Narasayya, "Automated selection of materialized views and indexes in SQL databases," in *Proceedings of the 2000 International Conference on Very Large Data Bases*, pp. 496–505, Cairo, Egypt, September 2000.

[24] S. Agrawal, S. Chaudhuri, and V. R. Narasayya, "Materialized view and index selection tool for Microsoft SQL Server 2000," in *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, p. 608, Santa Barbara, California, USA, June 2001.

[25] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra, "Matching events in a content-based subscription system," in *Proceedings of the 1999 ACM Symposium on Principles of Distributed Computing*, pp. 53–61, Atlanta, Georgia, USA, May 1999.

[26] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic, "DBToaster: Higher-order delta processing for dynamic, frequently fresh views," *Proceedings of the VLDB Endowment*, vol. 5, no. 10, pp. 968–979, 2012.

[27] A. Ailamaki, S. Babu, P. Furtado, S. Lightstone, G. M. Lohman, P. Martin, V. R. Narasayya, G. Pauley, K. Salem, K.-U. Sattler, and G. Weikum, "Report: 3rd International Workshop on Self-Managing Database Systems (SMDB 2008)," *IEEE Data Engineering Bulletin*, vol. 31, no. 4, pp. 2–5, 2008.

[28] A. Ailamaki, S. Chaudhuri, S. Lightstone, G. M. Lohman, P. Martin, K. Salem, and G. Weikum, "Report on the Second International Workshop on Self-Managing Database Systems (SMDB 2007)," *IEEE Data Engineering Bulletin*, vol. 30, no. 2, pp. 2–4, 2007.

[29] M. O. Akinde, O. G. Jensen, and M. H. Böhlen, "Minimizing detail data in data warehouses," in *Proceedings of the 1998 International Conference on Extending Database Technology*, pp. 293–307, Valencia, Spain, March 1998.

[30] M. Arenas, P. Barceló, L. Libkin, and F. Murlak, *Relational and XML Data Exchange*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[31] A. Arion, V. Benzaken, I. Manolescu, and Y. Papakonstantinou, "Structured materialized views for XML queries," in *Proceedings of the 2007 International Conference on Very Large Data Bases*, pp. 87–98, Vienna, Austria, September 2007.

[32] Z. Asgharzadeh Talebi, R. Chirkova, and Y. Fathi, "Exact and inexact methods for solving the problem of view selection for aggregate queries," *International Journal of Business Intelligence and Data Mining*, vol. 4, no. 3/4, pp. 391–415, 2009.

[33] Z. Asgharzadeh Talebi, R. Chirkova, Y. Fathi, and M. Stallmann, "Exact and inexact methods for selecting views and indexes for OLAP performance improvement," in *Proceedings of the 2008 International Conference on Extending Database Technology*, pp. 311–322, Nantes, France, March 2008.

[34] B. Babcock, M. Datar, and R. Motwani, "Load shedding in data stream systems," in Aggarwal [15], pp. 127–147.

[35] S. Babu, K. Munagala, J. Widom, and R. Motwani, "Adaptive caching for continuous queries," in *Proceedings of the 2005 International Conference on Data Engineering*, Tokyo, Japan, April 2005.

[36] S. Babu and K.-U. Sattler, "Report: 5th international workshop on self-managing database systems (SMDB 2010)," *IEEE Data Engineering Bulletin*, vol. 33, no. 3, pp. 4–7, 2010.

[37] S. Babu, U. Srivastava, and J. Widom, "Exploiting k-constraints to reduce memory overhead in continuous queries over data streams," *ACM Transactions on Database Systems*, vol. 29, no. 3, pp. 545–580, 2004.

[38] A. Balmin, F. Özcan, K. S. Beyer, R. Cochrane, and H. Pirahesh, "A framework for using materialized XPath views in XML query processing," in *Proceedings of the 2004 International Conference on Very Large Data Bases*, pp. 60–71, Toronto, Canada, August 2004.

[39] M. Bamha, F. Bentayeb, and G. Hains, "An efficient scalable parallel view maintenance algorithm for shared nothing multi-processor machines," in *Proceedings of the 1999 International Conference on Database and Expert Systems Applications*, pp. 616–625, Florence, Italy, August 1999.

[40] E. Baralis, S. Paraboschi, and E. Teniente, "Materialized views selection in a multidimensional database," in *Proceedings of the 1997 International Conference on Very Large Data Bases*, pp. 156–165, Athens, Greece, August 1997.

[41] P. Barceló, "Logical foundations of relational data exchange," *ACM SIGMOD Record*, vol. 38, no. 1, pp. 49–58, 2009.

[42] P. Belknap, B. Dageville, K. Dias, and K. Yagoub, "Self-tuning for SQL performance in Oracle Database 11g," in *Proceedings of the 2009 International Conference on Data Engineering*, pp. 1694–1700, Shanghai, China, March 2009.

[43] R. G. Bello, K. Dias, A. Downing, J. J. F. Jr., J. L. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin, "Materialized views in Oracle," in *Proceedings of the 1998 International Conference on Very Large Data Bases*, pp. 659–664, New York City, New York, USA, August 1998.

[44] M. Benedikt and G. Gottlob, "The impact of virtual views on containment," *Proceedings of the VLDB Endowment*, vol. 3, no. 1, pp. 297–308, 2010.

[45] P. A. Bernstein and L. M. Haas, "Information integration in the enterprise," *Communications of the ACM*, vol. 51, no. 9, pp. 72–79, 2008.

[46] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin, "Incoop: MapReduce for incremental computations," in *Proceedings of the 2011 ACM Symposium on Cloud Computing*, pp. 7:1–7:14, Cascais, Portugal, October 2011.

[47] P. Bizarro, S. Babu, D. DeWitt, and J. Widom, "Content-based routing: Different plans for different data," in *Proceedings of the 2005 International Conference on Very Large Data Bases*, Trondheim, Norway, August 2005.

[48] J. A. Blakeley, N. Coburn, and P.-Å. Larson, "Updating derived relations: Detecting irrelevant and autonomously computable updates," in *Proceedings of the 1986 International Conference on Very Large Data Bases*, pp. 457–466, Kyoto, Japan, August 1986.

[49] J. A. Blakeley, N. Coburn, and P.-V. Larson, "Updating derived relations: Detecting irrelevant and autonomously computable updates," *ACM Transactions on Database Systems*, vol. 14, no. 3, pp. 369–400, 1989.

[50] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa, "Efficiently updating materialized views," in *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pp. 61–71, Washington DC, USA, May 1986.

[51] J. A. Blakeley and N. L. Martin, "Join index, materialized view, and hybrid-hash join: A performance analysis," in *Proceedings of the 1990 International Conference on Data Engineering*, pp. 256–263, Los Angeles, California, USA, February 1990.

[52] A. Bonifati, M. H. Goodfellow, I. Manolescu, and D. Sileo, "Algebraic incremental maintenance of XML views," in *Proceedings of the 2011 International Conference on Extending Database Technology*, pp. 177–188, Uppsala, Sweden, March 2011.

[53] P. Bonnet and D. Shasha, "Index tuning," in Liu and Özsu [291], pp. 1433–1435.

[54] P. Bonnet and D. Shasha, "Schema tuning," in Liu and Özsu [291], pp. 2497–2499.

[55] N. Bruno and S. Chaudhuri, "Automatic physical database tuning: A relaxation-based approach," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pp. 227–238, Baltimore, Maryland, USA, June 2005.

[56] N. Bruno and S. Chaudhuri, "Physical design refinement: The "merge-reduce" approach," in *Proceedings of the 2006 International Conference on Extending Database Technology*, pp. 386–404, Munich, Germany, March 2006.

[57] N. Bruno and S. Chaudhuri, "To tune or not to tune? A lightweight physical design alerter," in *Proceedings of the 2006 International Conference on Very Large Data Bases*, pp. 499–510, Seoul, Korea, September 2006.

[58] N. Bruno and S. Chaudhuri, "Online approach to physical design tuning," in *Proceedings of the 2007 International Conference on Data Engineering*, pp. 826–835, Istanbul, Turkey, April 2007.

[59] N. Bruno and S. Chaudhuri, "Online AutoAdmin (physical design tuning)," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pp. 1067–1069, Beijing, China, June 2007.

[60] N. Bruno and S. Chaudhuri, "Physical design refinement: The merge-reduce approach," *ACM Transactions on Database Systems*, vol. 32, no. 4, pp. 28–43, 2007.

[61] N. Bruno and S. Chaudhuri, "Constrained physical design tuning," *Proceedings of the VLDB Endowment*, vol. 1, pp. 4–15, 2008.

[62] N. Bruno and S. Chaudhuri, "Constrained physical design tuning," *The VLDB Journal*, vol. 19, no. 1, pp. 21–44, 2010.

[63] N. Bruno, S. Chaudhuri, and G. Weikum, "Database tuning using online algorithms," in Liu and Özsu [291], pp. 741–744.

[64] P. Buneman and E. K. Clemons, "Efficiently monitoring relational databases," *ACM Transactions on Database Systems*, vol. 4, no. 3, pp. 368–382, 1979.

[65] C. J. Bunger, L. S. Colby, R. L. Cole, W. J. McKenna, G. Mulagund, and D. Wilhite, "Aggregate maintenance for data warehousing in Informix Red Brick Vista," in *Proceedings of the 2001 International Conference on Very Large Data Bases*, pp. 659–662, Roma, Italy, September 2001.

[66] A. Calì, D. Calvanese, G. D. Giacomo, and M. Lenzerini, "Data integration under integrity constraints," *Information Systems*, vol. 29, no. 2, pp. 147–163, 2004.

[67] D. Calvanese and G. D. Giacomo, "Data integration: A logic-based perspective," *AI Magazine*, vol. 26, no. 1, pp. 59–70, 2005.

[68] D. Calvanese, G. D. Giacomo, M. Lenzerini, D. Nardi, and R. Rosati, "Data integration in data warehousing," *International Journal of Cooperative Information Systems*, vol. 10, no. 3, pp. 237–271, 2001.

[69] D. Calvanese, G. D. Giacomo, M. Lenzerini, and R. Rosati, "Logical foundations of peer-to-peer data integration," in *Proceedings of the 2004 ACM Symposium on Principles of Database Systems*, pp. 241–251, Paris, France, June 2004.

[70] D. Calvanese, G. D. Giacomo, M. Lenzerini, and R. Rosati, "View-based query answering over description logic ontologies," in *Proceedings of the 2008 International Conference on Principles of Knowledge Representation and Reasoning*, pp. 242–251, Sydney, Australia, September 2008.

[71] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi, "Answering regular path queries using views," in *Proceedings of the 2000 International Conference on Data Engineering*, pp. 389–398, Los Angeles, California, USA, February 2000.

[72] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi, "Lossless regular views," in *Proceedings of the 2002 ACM Symposium on Principles of Database Systems*, pp. 247–258, Madison, Wisconsin, USA, June 2002.

[73] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi, "Query containment using views," in *Proceedings of the 2003 Italian Symposium on Advanced Database Systems*, pp. 467–474, Cetraro (CS), Italy, June 2003.

[74] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi, "View-based query containment," in *Proceedings of the 2003 ACM Symposium on Principles of Database Systems*, pp. 56–67, San Diego, California, USA, June 2003.

[75] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi, "View-based query processing: On the relationship between rewriting, answering and losslessness," in *Proceedings of the 2005 International Conference on Database Theory*, pp. 321–336, Edinburgh, UK, Janaury 2005.

[76] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi, "View-based query processing: On the relationship between rewriting, answering and losslessness," *Theoretical Computer Science*, vol. 371, no. 3, pp. 169–182, 2007.

[77] K. S. Candan, D. Agrawal, W.-S. Li, O. Po, and W.-P. Hsiung, "View invalidation for dynamic content caching in multitiered architectures," in *Proceedings of the 2002 International Conference on Very Large Data Bases*, pp. 562–573, Hong Kong, China, September 2002.

[78] S. Castano, V. D. Antonellis, and S. D. C. di Vimercati, "Global viewing of heterogeneous data sources," *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, no. 2, pp. 277–297, 2001.

[79] B. Cautis, A. Deutsch, and N. Onose, "XPath rewriting using multiple views: Achieving completeness and efficiency," in *Proceedings of the 2008 International Workshop on the Web and Databases*, Vancouver, Canada, June 2008.

[80] B. Cautis, A. Deutsch, and N. Onose, "Querying data sources that export infinite sets of views," in *Proceedings of the 2009 International Conference on Database Theory*, pp. 84–97, Saint-Petersburg, Russia, March 2009.

[81] B. Cautis, A. Deutsch, N. Onose, and V. Vassalos, "Efficient rewriting of XPath queries using query set specifications," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 301–312, 2009.

[82] B. Cautis, A. Deutsch, N. Onose, and V. Vassalos, "Querying XML data sources that export very large sets of views," *ACM Transactions on Database Systems*, vol. 36, no. 1, p. 5, 2011.

[83] S. Ceri and J. Widom, "Deriving production rules for incremental view maintenance," in *Proceedings of the 1991 International Conference on Very Large Data Bases*, pp. 577–589, Barcelona, Catalonia, Spain, 1991.

[84] A. Chandra and P. Merlin, "Optimal implementation of conjunctive queries in relational data bases," in *Proceedings of the 1977 ACM Symposium on Theory of Computing*, pp. 77–90, Boulder, Colorado, USA, May 1977.

[85] S. Chandrasekaran and M. J. Franklin, "PSoup: A system for streaming queries over streaming data," *The VLDB Journal*, vol. 12, no. 2, pp. 140–156, 2003.

[86] S. Chaudhuri, "An overview of query optimization in relational systems," in *Proceedings of the 1998 ACM Symposium on Principles of Database Systems*, pp. 34–43, Seattle, Washington, USA, June 1998.

[87] S. Chaudhuri, E. Christensen, G. Graefe, V. R. Narasayya, and M. J. Zwilling, "Self-tuning technology in Microsoft SQL Server," *IEEE Data Engineering Bulletin*, vol. 22, no. 2, pp. 20–26, 1999.

[88] S. Chaudhuri, M. Datar, and V. R. Narasayya, "Index selection for databases: A hardness study and principled heuristic solution," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, pp. 1313–1323, 2004.

[89] S. Chaudhuri and U. Dayal, "An overview of data warehousing and OLAP technology," *ACM SIGMOD Record*, vol. 26, no. 1, pp. 65–74, 1997.

[90] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim, "Optimizing queries with materialized views," in *Proceedings of the 1995 International Conference on Data Engineering*, pp. 190–200, Taipei, Taiwan, March 1995.

[91] S. Chaudhuri and V. R. Narasayya, "An efficient cost-driven index selection tool for Microsoft SQL server," in *Proceedings of the 1997 International Conference on Very Large Data Bases*, pp. 146–155, Athens, Greece, August 1997.

[92] S. Chaudhuri and V. R. Narasayya, "AutoAdmin 'what-if' index analysis utility," in *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pp. 367–378, Seattle, Washington, USA, May 1998.

[93] S. Chaudhuri and V. R. Narasayya, "Self-tuning database systems: A decade of progress," in *Proceedings of the 2007 International Conference on Very Large Data Bases*, pp. 3–14, Vienna, Austria, September 2007.

[94] S. Chaudhuri, V. R. Narasayya, and G. Weikum, "Database tuning using combinatorial search," in Liu and Özsu [291], pp. 738–741.

[95] S. Chaudhuri and M. Y. Vardi, "Optimization of *real* conjunctive queries," in *Proceedings of the 1993 ACM Symposium on Principles of Database Systems*, pp. 59–70, Washington DC, USA, May 1993.

[96] S. Chaudhuri and G. Weikum, "Self-management technology in databases," in Liu and Özsu [291], pp. 2550–2555.

[97] S. Chaudhuri and G. Weikum, "Rethinking database system architecture: Towards a self-tuning risc-style database system," in *Proceedings of the 2000*

International Conference on Very Large Data Bases, pp. 1–10, Cairo, Egypt, September 2000.

[98] L. W. F. Chaves, E. Buchmann, F. Hueske, and K. Böhm, "Towards materialized view selection for distributed databases," in *Proceedings of the 2009 International Conference on Extending Database Technology*, pp. 1088–1099, Saint Petersburg, Russia, March 2009.

[99] D. Chen and C.-Y. Chan, "ViewJoin: Efficient view-based evaluation of tree pattern queries," in *Proceedings of the 2010 International Conference on Data Engineering*, pp. 816–827, Long Beach, California, USA, March 2010.

[100] J. Chen, S. Chen, and E. A. Rundensteiner, "A transactional model for data warehouse maintenance," in *Proceedings of the 2002 International Conference on Conceptual Modeling*, pp. 247–262, Tampere, Finland, October 2002.

[101] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, "NiagaraCQ: A scalable continuous query system for internet databases," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pp. 379–390, Dallas, Texas, USA, May 2000.

[102] J. Chen, X. Zhang, S. Chen, A. Koeller, and E. A. Rundensteiner, "DyDa: Data warehouse maintenance in fully concurrent environments," in *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, p. 619, Santa Barbara, California, USA, June 2001.

[103] L. Chen and E. A. Rundensteiner, "XCache: XQuery-based caching system," in *Proceedings of the 2002 International Workshop on the Web and Databases*, pp. 31–36, Madison, Wisconsin, USA, June 2002.

[104] S. Chen, B. Liu, and E. A. Rundensteiner, "Multiversion-based view maintenance over distributed data sources," *ACM Transactions on Database Systems*, vol. 29, no. 4, pp. 675–709, 2004.

[105] J. Cheney, L. Chiticariu, and W. C. Tan, "Provenance in databases: Why, how and where," *Foundations and Trends in Databases*, vol. 1, no. 4, pp. 379–474, 2009.

[106] R. Chirkova, "Query containment," in Liu and Özsu [291], pp. 2249–2253.

[107] R. Chirkova, "The view-selection problem has an exponential-time lower bound for conjunctive queries and views," in *Proceedings of the 2002 ACM Symposium on Principles of Database Systems*, pp. 159–168, Madison, Wisconsin, USA, June 2002.

[108] R. Chirkova, A. Y. Halevy, and D. Suciu, "A formal perspective on the view selection problem," *The VLDB Journal*, vol. 11, no. 3, pp. 216–237, 2002.

[109] R. Chirkova and C. Li, "Materializing views with minimal size to answer queries," in *Proceedings of the 2003 ACM Symposium on Principles of Database Systems*, pp. 38–48, San Diego, California, USA, June 2003.

[110] R. Chirkova, C. Li, and J. Li, "Answering queries using materialized views with minimum size," *The VLDB Journal*, vol. 15, no. 3, pp. 191–210, 2006.

[111] S. Cohen, "Aggregation: Expressiveness and containment," in Liu and Özsu [291], pp. 59–63.

[112] S. Cohen, "Equivalence of queries combining set and bag-set semantics," in *Proceedings of the 2006 ACM Symposium on Principles of Database Systems*, pp. 70–79, Chicago, Illinois, USA, June 2006.

[113] S. Cohen, "User-defined aggregate functions: Bridging theory and practice," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pp. 49–60, Chicago, Illinois, USA, June 2006.

[114] S. Cohen, "Equivalence of queries that are sensitive to multiplicities," *The VLDB Journal*, vol. 18, pp. 765–785, 2009.

[115] S. Cohen, W. Nutt, and Y. Sagiv, "Rewriting queries with arbitrary aggregation functions using views," *ACM Transactions on Database Systems*, vol. 31, no. 2, pp. 672–715, 2006.

[116] S. Cohen, W. Nutt, and Y. Sagiv, "Deciding equivalences among conjunctive aggregate queries," *Journal of the ACM*, vol. 54, no. 2, 2007.

[117] S. Cohen, W. Nutt, and A. Serebrenik, "Rewriting aggregate queries using views," in *Proceedings of the 1999 ACM Symposium on Principles of Database Systems*, pp. 155–166, Philadelphia, Pennsylvania, USA, June 1999.

[118] S. Cohen, W. Nutt, and A. Serebrenik, "Algorithms for rewriting aggregate queries using views," in *Proceedings of the 2000 East European Conference on Advances in Databases and Information Systems Held Jointly with the International Conference on Database Systems for Advanced Applications*, pp. 65–78, Prague, Czech Republic, September 2000.

[119] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey, "Algorithms for deferred view maintenance," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pp. 469–480, Montreal, Quebec, Canada, June 1996.

[120] L. S. Colby, A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross, "Supporting multiple view maintenance policies," in *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pp. 405–416, Tucson, Arizona, USA, May 1997.

[121] M. Compton, "Finding equivalent rewritings with exact views," in *Proceedings of the 2009 International Conference on Data Engineering*, pp. 1243–1246, Shanghai, China, March 2009.

[122] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online," in *Proceedings of the 2010 USENIX Symposium on Networked Systems Design and Implementation*, San Jose, California, USA, April 2010.

[123] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine, "Synopses for massive data: Samples, histograms, wavelets, sketches," *Foundations and Trends in Databases*, vol. 4, no. 1–3, pp. 1–294, 2012.

[124] Y. Cui and J. Widom, "Storing auxiliary data for efficient maintenance and lineage tracing of complex views," in *Proceedings of the 2000 International Workshop on Design and Management of Data Warehouses*, Stockholm, Sweden, June 2000.

[125] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zaït, and M. Ziauddin, "Automatic SQL tuning in Oracle 10g," in *Proceedings of the 2004 International Conference on Very Large Data Bases*, pp. 1098–1109, Toronto, Canada, August 2004.

[126] B. Dageville and K. Dias, "Oracle's self-tuning architecture and solutions," *IEEE Data Engineering Bulletin*, vol. 29, no. 3, pp. 24–31, 2006.

[127] N. N. Dalvi, C. Re, and D. Suciu, "Queries and materialized views on probabilistic databases," *Journal of Computer and System Sciences*, vol. 77, no. 3, pp. 473–490, 2011.

[128] N. N. Dalvi and D. Suciu, "Answering queries from statistics and probabilistic views," in *Proceedings of the 2005 International Conference on Very Large Data Bases*, pp. 805–816, Trondheim, Norway, August 2005.

[129] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan, "Semantic data caching and replacement," in *Proceedings of the 1996 International Conference on Very Large Data Bases*, pp. 330–341, Mumbai (Bombay), India, September 1996.

[130] J. Dean and S. Ghemawat, "MapReduce: A flexible data processing tool," *Communications of the ACM*, vol. 53, no. 1, pp. 72–77, 2010.

[131] D. DeHaan, P.-Å. Larson, and J. Zhou, "Stacked indexed views in Microsoft SQL Server," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pp. 179–190, Baltimore, Maryland, USA, June 2005.

[132] A. Deligiannakis, "View maintenance aspects," in Liu and Özsu [291], pp. 3328–3331.

[133] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White, "Cayuga: A general purpose event monitoring system," in *Proceedings of the 2007 Conference on Innovative Data Systems Research*, pp. 412–422, Asilomar, California, USA, January 2007.

[134] A. Deutsch, "FOL modeling of integrity constraints (dependencies)," in Liu and Özsu [291], pp. 1155–1161.

[135] A. Deutsch, Y. Katsis, and Y. Papakonstantinou, "Determining source contribution in integration systems," in *Proceedings of the 2005 ACM Symposium on Principles of Database Systems*, pp. 304–315, Baltimore, Maryland, USA, June 2005.

[136] A. Deutsch, B. Ludäscher, and A. Nash, "Rewriting queries using views with access patterns under integrity constraints," in *Proceedings of the 2005 International Conference on Database Theory*, pp. 352–367, Edinburgh, UK, January 2005.

[137] A. Deutsch and A. Nash, "Chase," in Liu and Özsu [291], pp. 323–327.

[138] A. Deutsch, L. Popa, and V. Tannen, "Query reformulation with constraints," *ACM SIGMOD Record*, vol. 35, no. 1, pp. 65–73, 2006.

[139] A. Doan, A. Halevy, and Z. Ives, *Principles of Data Integration*. Morgan Kaufmann, 1st ed., July 2012.

[140] A. Doan and A. Y. Halevy, "Semantic integration research in the database community: A brief survey," *AI Magazine*, vol. 26, no. 1, pp. 83–94, 2005.

[141] G. Dong and J. Su, "Incremental computation of queries," in Liu and Özsu [291], pp. 1414–1417.

[142] A. El-Helw, I. F. Ilyas, and C. Zuzarte, "Statadvisor: Recommending statistical views," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1306–1317, 2009.

[143] M. El-Sayed, E. A. Rundensteiner, and M. Mani, "Incremental maintenance of materialized XQuery views," in *Proceedings of the 2006 International Conference on Data Engineering*, p. 129, Atlanta, Georgia, USA, April 2006.

[144] C. Elkan, "Independence of logic database queries and updates," in *Proceedings of the 1990 ACM Symposium on Principles of Database Systems*, pp. 154–160, Nashville, Tennessee, USA, April 1990.

[145] H. Engström, S. Chakravarthy, and B. Lings, "A systematic approach to selecting maintenance policies in a data warehouse environment," in *Proceedings of the 2002 International Conference on Extending Database Technology*, pp. 317–335, Prague, Czech Republic, March 2002.

[146] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys*, vol. 35, no. 2, pp. 114–131, 2003.

[147] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha, "Filtering algorithms and implementation for very fast publish/subscribe," in *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, Santa Barbara, California, USA, June 2001.

[148] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis, "Rewriting regular XPath queries on XML views," in *Proceedings of the 2007 International Conference on Data Engineering*, pp. 666–675, Istanbul, Turkey, April 2007.

[149] S. J. Finkelstein, M. Schkolnick, and P. Tiberio, "Physical database design for relational databases," *ACM Transactions on Database Systems*, vol. 13, no. 1, 1988.

[150] S. Flesca and S. Greco, "Rewriting queries using views," *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, no. 6, pp. 980–995, 2001.

[151] N. Folkert, A. Gupta, A. Witkowski, S. Subramanian, S. Bellamkonda, S. Shankar, T. Bozkaya, and L. Sheng, "Optimizing refresh of a set of materialized views," in *Proceedings of the 2005 International Conference on Very Large Data Bases*, pp. 1043–1054, Trondheim, Norway, August 2005.

[152] Y. Fu, K. Kowalczykowski, K. W. Ong, Y. Papakonstantinou, and K. K. Zhao, "Ajax-based report pages as incrementally rendered views," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pp. 567–578, Indianapolis, Indiana, USA, June 2010.

[153] Y. Fu, K. W. Ong, Y. Papakonstantinou, and M. Petropoulos, "The SQL-based all-declarative FORWARD web application development framework," in *Proceedings of the 2011 Conference on Innovative Data Systems Research*, pp. 69–78, Asilomar, California, USA, January 2011.

[154] A. Fuxman, P. G. Kolaitis, R. J. Miller, and W. C. Tan, "Peer data exchange," in *Proceedings of the 2005 ACM Symposium on Principles of Database Systems*, pp. 160–171, Baltimore, Maryland, USA, June 2005.

[155] A. Fuxman, P. G. Kolaitis, R. J. Miller, and W. C. Tan, "Peer data exchange," *ACM Transactions on Database Systems*, vol. 31, no. 4, pp. 1454–1498, 2006.

[156] A. Fuxman and R. J. Miller, "First-order query rewriting for inconsistent databases," in *Proceedings of the 2005 International Conference on Database Theory*, pp. 337–351, Edinburgh, UK, January 2005.

[157] H. Garcia-Molina, W. J. Labio, and J. Yang, "Expiring data in a warehouse," in *Proceedings of the 1998 International Conference on Very Large Data Bases*, pp. 500–511, New York City, New York, USA, August 1998.

[158] H. Garcia-Molina, J. Ullman, and J. Widom, *Database Systems: The Complete Book*. Pearson Prentice Hall, 2009.

[159] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database Systems: The Complete Book*. Pearson Education, 2nd ed., 2009.

[160] K. E. Gebaly and A. Aboulnaga, "Robustness in automatic physical database design," in *Proceedings of the 2008 International Conference on Extending Database Technology*, pp. 145–156, Nantes, France, March 2008.

[161] R. Gemulla and W. Lehner, "Deferred maintenance of disk-based random samples," in *Proceedings of the 2006 International Conference on Extending Database Technology*, pp. 423–441, Munich, Germany, March 2006.

[162] M. R. Genesereth, *Data Integration: The Relational Logic Approach. Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool Publishers, 2010.

[163] T. M. Ghanem, A. K. Elmagarmid, P.-Å. Larson, and W. G. Aref, "Supporting views in data stream management systems," *ACM Transactions on Database Systems*, vol. 35, no. 1, 2010.

[164] G. D. Giacomo, D. Lembo, M. Lenzerini, and R. Rosati, "On reconciling data exchange, data integration, and peer data management," in *Proceedings of the 2007 ACM Symposium on Principles of Database Systems*, pp. 133–142, Beijing, China, June 2007.

[165] P. Godfrey and J. Gryz, "View disassembly: A rewrite that extracts portions of views," *Journal of Computer and System Sciences*, vol. 73, no. 6, pp. 941–961, 2007.

[166] P. Godfrey, J. Gryz, A. Hoppe, W. Ma, and C. Zuzarte, "Query rewrites with views for XML in DB2," in *Proceedings of the 2009 International Conference on Data Engineering*, pp. 1339–1350, Shanghai, China, March 2009.

[167] J. Goldstein and P.-Å. Larson, "Optimizing queries using materialized views: A practical, scalable solution," in *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pp. 331–342, Santa Barbara, California, USA, June 2001.

[168] G. Gou, M. Kormilitsin, and R. Chirkova, "Query evaluation using overlapping views: Completeness and efficiency," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pp. 37–48, Chicago, Illinois, USA, June 2006.

[169] G. Graefe and H. A. Kuno, "Self-selecting, self-tuning, incrementally optimized indexes," in *Proceedings of the 2010 International Conference on Extending Database Technology*, pp. 371–381, Lausanne, Switzerland, March 2010.

[170] G. Graefe and M. J. Zwilling, "Transaction support for indexed views," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, Paris, France, June 2004.

[171] G. Grahne and A. Thomo, "Query containment and rewriting using views for regular path queries under constraints," in *Proceedings of the 2003 ACM Symposium on Principles of Database Systems*, pp. 111–122, San Diego, California, USA, June 2003.

[172] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total," in *Proceedings of the 1996 International Conference on Data Engineering*, pp. 152–159, New Orleans, Louisiana, USA, February 1996.

[173] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, and M. Venkatrao, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data Mining and Knowledge Discovery*, vol. 1, no. 1, pp. 29–53, 1997.

[174] T. J. Green, "Bag semantics," in Liu and Özsu [291], pp. 201–206.

[175] T. J. Green and Z. G. Ives, "Recomputing materialized instances after changes to mappings and data," in *Proceedings of the 2012 International Conference on Data Engineering*, pp. 330–341, Washington DC, USA, April 2012.

[176] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen, "Update exchange with mappings and provenance," in *Proceedings of the 2007 International Conference on Very Large Data Bases*, pp. 675–686, Vienna, Austria, September 2007.

[177] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen, "Provenance in ORCHESTRA," *IEEE Data Engineering Bulletin*, vol. 33, no. 3, pp. 9–16, 2010.

[178] T. J. Green, G. Karvounarakis, N. E. Taylor, O. Biton, Z. G. Ives, and V. Tannen, "ORCHESTRA: Facilitating collaborative data sharing," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pp. 1131–1133, Beijing, China, June 2007.

[179] T. Griffin and B. Kumar, "Algebraic change propagation for semijoin and outerjoin queries," *ACM SIGMOD Record*, vol. 27, no. 3, pp. 22–27, 1998.

[180] T. Griffin and L. Libkin, "Incremental maintenance of views with duplicates," in *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pp. 328–339, San Jose, California, USA, May 1995.

[181] T. Griffin, L. Libkin, and H. Trickey, "An improved algorithm for the incremental recomputation of active relational expressions," *IEEE Transactions on Knowledge and Data Engineering*, vol. 9, no. 3, pp. 508–511, 1997.

[182] S. Grumbach, M. Rafanelli, and L. Tininini, "On the equivalence and rewriting of aggregate queries," *Acta Informatica*, vol. 40, no. 8, pp. 529–584, 2004.

[183] S. Grumbach and L. Tininini, "On the content of materialized aggregate views," *Journal of Computer and System Sciences*, vol. 66, no. 1, pp. 133–168, 2003.

[184] A. Gupta and J. A. Blakeley, "Using partial information to update materialized views," *Information Systems*, vol. 20, no. 8, pp. 641–662, 1995.

[185] A. Gupta, V. Harinarayan, and D. Quass, "Aggregate-query processing in data warehousing environments," in *Proceedings of the 1995 International Conference on Very Large Data Bases*, pp. 358–369, Zurich, Switzerland, September 1995.

[186] A. Gupta, H. V. Jagadish, and I. S. Mumick, "Data integration using self-maintainable views," in *Proceedings of the 1996 International Conference on Extending Database Technology*, pp. 140–144, Avignon, France, March 1996.

[187] A. Gupta and I. S. Mumick, "Maintenance of materialized views: Problems, techniques, and applications," *IEEE Data Engineering Bulletin*, vol. 18, no. 2, pp. 3–18, 1995.

[188] A. Gupta and I. S. Mumick, eds., *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, 1999.

[189] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, "Maintaining views incrementally," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pp. 157–166, Washington DC, USA, May 1993.

[190] A. K. Gupta, A. Y. Halevy, and D. Suciu, "View selection for stream processing," in *Proceedings of the 2002 International Workshop on the Web and Databases*, pp. 83–88, Madison, Wisconsin, USA, June 2002.

[191] A. K. Gupta, D. Suciu, and A. Y. Halevy, "The view selection problem for XML content based routing," in *Proceedings of the 2003 ACM Symposium on Principles of Database Systems*, pp. 68–77, San Diego, California, USA, June 2003.

[192] H. Gupta, "Selection of views to materialize in a data warehouse," in *Proceedings of the 1997 International Conference on Database Theory*, pp. 98–112, Delphi, Greece, January 1997.

[193] H. Gupta, "Selection and maintenance of views in a data warehouse," PhD thesis, Department of Computer Science, Stanford University, 1999.

[194] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman, "Index selection for OLAP," in *Proceedings of the 1997 International Conference on Data Engineering*, pp. 208–219, Birmingham, UK, April 1997.

[195] H. Gupta and I. S. Mumick, "Selection of views to materialize under a maintenance cost constraint," in *Proceedings of the 1999 International Conference on Database Theory*, pp. 453–470, Jerusalem, Israel, January 1999.

[196] H. Gupta and I. S. Mumick, "Selection of views to materialize in a data warehouse," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 1, pp. 24–43, 2005.

[197] H. Gupta and I. S. Mumick, "Incremental maintenance of aggregate and outerjoin expressions," *Information Systems*, vol. 31, no. 6, pp. 435–464, 2006.

[198] N. Gupta, L. Kot, G. Bender, S. Roy, J. Gehrke, and C. Koch, "Coordination through querying in the Youtopia system," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pp. 1331–1334, Athens, Greece, June 2011.

[199] L. M. Haas, "Beauty and the beast: The theory and practice of information integration," in *Proceedings of the 2007 International Conference on Database Theory*, pp. 28–43, Barcelona, Spain, January 2007.

[200] A. Halevy, "Data integration: A status report," in *Datenbanksysteme für Business, Technologie und Web*, pp. 24–29, Leipzig, Germany, February 2003.

[201] A. Y. Halevy, "Information integration," in Liu and Özsu [291], pp. 1490–1496.

[202] A. Y. Halevy, "Theory of answering queries using views," *ACM SIGMOD Record*, vol. 29, no. 4, pp. 40–47, 2000.

[203] A. Y. Halevy, "Answering queries using views: A survey," *The VLDB Journal*, vol. 10, no. 4, pp. 270–294, 2001.

[204] A. Y. Halevy, N. Ashish, D. Bitton, M. J. Carey, D. Draper, J. Pollock, A. Rosenthal, and V. Sikka, "Enterprise information integration: Successes, challenges and controversies," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pp. 778–787, Baltimore, Maryland, USA, June 2005.

[205] A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov, "The Piazza peer data management system," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 7, pp. 787–798, 2004.

[206] A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov, "Piazza: Data management infrastructure for semantic web applications," in *Proceedings of the 2003 International Conference on World Wide Web*, pp. 556–567, Budapest, Hungary, May 2003.

[207] A. Y. Halevy, A. Rajaraman, and J. J. Ordille, "Data integration: The teenage years," in *Proceedings of the 2006 International Conference on Very Large Data Bases*, pp. 9–16, Seoul, Korea, September 2006.

[208] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 3rd ed., 2005.

[209] E. N. Hanson, "A performance analysis of view materialization strategies," in *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pp. 440–453, San Francisco, California, USA, May 1987.

[210] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon, "Scalable trigger processing," in *Proceedings of the 1999 International Conference on Data Engineering*, pp. 266–275, Sydney, Austrialia, March 1999.

[211] N. Hanusse, S. Maabout, and R. Tofan, "A view selection algorithm with performance guarantee," in *Proceedings of the 2009 International Conference on Extending Database Technology*, pp. 946–957, Saint Petersburg, Russia, March 2009.

[212] V. Harinarayan, A. Rajaraman, and J. D. Ullman, "Implementing data cubes efficiently," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pp. 205–216, Montreal, Quebec, Canada, June 1996.

[213] H. He, J. Xie, J. Yang, and H. Yu, "Asymmetric batch incremental view maintenance," in *Proceedings of the 2005 International Conference on Data Engineering*, pp. 106–117, Tokyo, Japan, April 2005.

[214] J. M. Hellerstein, M. Stonebraker, and J. R. Hamilton, "Architecture of a database system," *Foundations and Trends in Databases*, vol. 1, no. 2, pp. 141–259, 2007.

[215] K. Hose, D. Klan, and K.-U. Sattler, "Online tuning of aggregation tables for OLAP," in *Proceedings of the 2009 International Conference on Data Engineering*, pp. 1679–1686, Shanghai, China, March 2009.

[216] V. Hristidis and M. Petropoulos, "Semantic caching of XML databases," in *Proceedings of the 2002 International Workshop on the Web and Databases*, pp. 25–30, Madison, Wisconsin, USA, June 2002.

[217] R. Hull and G. Zhou, "A framework for supporting data integration using the materialized and virtual approaches," in *Proceedings of the 1996 ACM*

SIGMOD International Conference on Management of Data, pp. 481–492, Montreal, Quebec, Canada, June 1996.

[218] E. Hung, Y. Deng, and V. S. Subrahmanian, "RDF aggregate queries and views," in *Proceedings of the 2005 International Conference on Data Engineering*, pp. 717–728, Tokyo, Japan, April 2005.

[219] C. A. Hurtado, C. Gutiérrez, and A. O. Mendelzon, "Capturing summarizability with integrity constraints in OLAP," *ACM Transactions on Database Systems*, vol. 30, no. 3, pp. 854–886, 2005.

[220] N. Huyn, "Efficient view self-maintenance," in *Proceedings of the 1996 Workshop on Materialized Views*, pp. 17–25, 1996.

[221] N. Huyn, "Multiple-view self-maintenance in data warehousing environments," in *Proceedings of the 1997 International Conference on Very Large Data Bases*, pp. 26–35, Athens, Greece, August 1997.

[222] N. Huyn, "Speeding up view maintenance using cheap filters at the warehouse," in *Proceedings of the 2000 International Conference on Data Engineering*, p. 308, Los Angeles, California, USA, February 2000.

[223] Y. Ioannidis and R. Ramakrishnan, "Containment of conjunctive queries: Beyond relations as sets," *ACM Transactions on Database Systems*, vol. 20, no. 3, pp. 288–324, 1995.

[224] Y. E. Ioannidis, "Query optimization," in *The Computer Science and Engineering Handbook*, (A. B. Tucker, ed.), pp. 1038–1057, CRC Press, 1997.

[225] Z. G. Ives, T. J. Green, G. Karvounarakis, N. E. Taylor, V. Tannen, P. P. Talukdar, M. Jacob, and F. Pereira, "The ORCHESTRA collaborative data sharing system," *ACM SIGMOD Record*, vol. 37, no. 3, pp. 26–32, 2008.

[226] Z. G. Ives, A. Y. Halevy, P. Mork, and I. Tatarinov, "Piazza: Mediation and integration infrastructure for semantic web data," *The Journal of Web Semantics*, vol. 1, no. 2, pp. 155–175, 2004.

[227] Z. G. Ives, N. Khandelwal, A. Kapur, and M. Cakir, "ORCHESTRA: Rapid, collaborative sharing of dynamic data," in *Proceedings of the 2005 Conference on Innovative Data Systems Research*, pp. 107–118, Asilomar, California, USA, January 2005.

[228] H. V. Jagadish, I. S. Mumick, and A. Silberschatz, "View maintenance issues for the chronicle data model," in *Proceedings of the 1995 ACM Symposium on Principles of Database Systems*, pp. 113–124, San Jose, California, USA, June 1995.

[229] M. Jarke and J. Koch, "Query optimization in database systems," *ACM Computing Surveys*, vol. 16, no. 2, pp. 111–152, 1984.

[230] C. Jermaine, A. Pol, and S. Arumugam, "Online maintenance of very large random samples," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pp. 299–310, Paris, France, June 2004.

[231] H. Jiang, D. Gao, and W.-S. Li, "Exploiting correlation and parallelism of materialized-view recommendation for distributed data warehouses," in *Proceedings of the 2007 International Conference on Data Engineering*, pp. 276–285, Istanbul, Turkey, April 2007.

[232] S. Joshi and C. Jermaine, "Materialized sample views for database approximation," in *Proceedings of the 2006 International Conference on Data Engineering*, p. 151, Atlanta, Georgia, USA, April 2006.

[233] S. Joshi and C. M. Jermaine, "Materialized sample views for database approximation," *IEEE Transactions on Knowledge and Data Engineering*, vol. 20, no. 3, pp. 337–351, 2008.

[234] B. Kähler and O. Risnes, "Extending logging for database snapshot refresh," in *Proceedings of the 1987 International Conference on Very Large Data Bases*, pp. 389–398, Brighton, England, September 1987.

[235] H.-G. Kang and C.-W. Chung, "Exploiting versions for on-line data warehouse maintenance in MOLAP servers," in *Proceedings of the 2002 International Conference on Very Large Data Bases*, pp. 742–753, Hong Kong, China, September 2002.

[236] J. Kang, J. F. Naughton, and S. Viglas, "Evaluating window joins over unbounded streams," in *Proceedings of the 2003 International Conference on Data Engineering*, pp. 341–352, Bangalore, India, March 2003.

[237] H. J. Karloff and M. Mihail, "On the complexity of the view-selection problem," in *Proceedings of the 1999 ACM Symposium on Principles of Database Systems*, pp. 167–173, Philadelphia, Pennsylvania, USA, June 1999.

[238] G. Karvounarakis and Z. G. Ives, "Bidirectional mappings for data and update exchange," in *Proceedings of the 2008 International Workshop on the Web and Databases*, Vancouver, Canada, June 2008.

[239] Y. Katsis and Y. Papakonstantinou, "View-based data integration," in Liu and Özsu [291], pp. 3332–3339.

[240] A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, D. Quass, and K. A. Ross, "Concurrency control theory for deferred materialized views," in *Proceedings of the 1997 International Conference on Database Theory*, pp. 306–320, Delphi, Greece, January 1997.

[241] O. Kennedy, Y. Ahmad, and C. Koch, "DBToaster: Agile views for a dynamic data management system," in *Proceedings of the 2011 Conference on Innovative Data Systems Research*, pp. 284–295, Asilomar, California, USA, January 2011.

[242] R. Kimball and M. Ross, *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses*. John Wiley, 2nd ed., 2002.

[243] H. Kimura, G. Huo, A. Rasin, S. Madden, and S. B. Zdonik, "CORADD: Correlation aware database designer for materialized views and indexes," *Proceedings of the VLDB Endowment*, vol. 3, no. 1, pp. 1103–1113, 2010.

[244] A. Klug, "On conjunctive queries containing inequalities," *Journal of the ACM*, vol. 35, no. 1, pp. 146–160, 1988.

[245] P. G. Kolaitis, D. L. Martin, and M. N. Thakur, "On the complexity of the containment problem for conjunctive queries with built-in predicates," in *Proceedings of the 1998 ACM Symposium on Principles of Database Systems*, pp. 197–204, Seattle, Washington, USA, June 1998.

[246] M. Kormilitsin, R. Chirkova, Y. Fathi, and M. Stallmann, "View and index selection for query-performance improvement: Quality-centered algorithms and heuristics," in *Proceedings of the 2008 International Conference on Information and Knowledge Management*, pp. 1329–1330, Napa Valley, California, USA, October 2008.

[247] M. Kormilitsin, R. Chirkova, Y. Fathi, and M. Stallmann, "Systematic exploration of efficient query plans for automated database restructuring," in

Proceedings of the 2009 East European Conference on Advances in Databases and Information Systems, pp. 133–148, Riga, Latvia, September 2009.

[248] L. Kot and C. Koch, "Cooperative update exchange in the Youtopia system," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 193–204, 2009.

[249] Y. Kotidis and N. Roussopoulos, "DynaMat: A dynamic view management system for data warehouses," in *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pp. 371–382, Philadelphia, Pennsylvania, USA, May 1999.

[250] Y. Kotidis and N. Roussopoulos, "A case for dynamic view management," *ACM Transactions on Database Systems*, vol. 26, no. 4, pp. 388–423, 2001.

[251] S. Kulkarni and M. K. Mohania, "Concurrent maintenance of views using multiple versions," in *Proceedings of the 1999 International Database Engineering and Applications Symposium*, pp. 254–259, Montreal, Canada, August 1999.

[252] W. Labio, D. Quass, and B. Adelberg, "Physical database design for data warehouses," in *Proceedings of the 1997 International Conference on Data Engineering*, pp. 277–288, Birmingham, UK, April 1997.

[253] W. J. Labio, J. Yang, Y. Cui, H. Garcia-Molina, and J. Widom, "Performance issues in incremental warehouse maintenance," in *Proceedings of the 2000 International Conference on Very Large Data Bases*, pp. 461–472, Cairo, Egypt, September 2000.

[254] W. J. Labio, R. Yerneni, and H. Garcia-Molina, "Shrinking the warehouse update window," in *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pp. 383–394, Philadelphia, Pennsylvania, USA, May 1999.

[255] A. Labrinidis, Q. Luo, J. Xu, and W. Xue, "Caching and materialization for web databases," *Foundations and Trends in Databases*, vol. 2, no. 3, pp. 169–266, 2009.

[256] L. V. S. Lakshmanan, J. Pei, and Y. Zhao, "QC-trees: An efficient summary structure for semantic OLAP," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pp. 64–75, San Diego, California, USA, June 2003.

[257] L. V. S. Lakshmanan, W. H. Wang, and Z. J. Zhao, "Answering tree pattern queries using views," in *Proceedings of the 2006 International Conference on Very Large Data Bases*, pp. 571–582, Seoul, Korea, September 2006.

[258] P.-Å. Larson, W. Lehner, J. Zhou, and P. Zabback, "Cardinality estimation using sample views with quality assurance," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pp. 175–186, Beijing, China, June 2007.

[259] P.-Å. Larson, W. Lehner, J. Zhou, and P. Zabback, "Exploiting self-monitoring sample views for cardinality estimation," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pp. 1073–1075, Beijing, China, June 2007.

[260] P.-Å. Larson and H. Z. Yang, "Computing queries from derived relations," in *Proceedings of the 1985 International Conference on Very Large Data Bases*, pp. 259–269, Stockholm, Sweden, August 1985.

[261] P.-Å. Larson and J. Zhou, "View matching for outer-join views," in *Proceedings of the 2005 International Conference on Very Large Data Bases*, pp. 445–456, Trondheim, Norway, August 2005.

[262] P.-Å. Larson and J. Zhou, "Efficient maintenance of materialized outer-join views," in *Proceedings of the 2007 International Conference on Data Engineering*, pp. 56–65, Istanbul, Turkey, April 2007.

[263] P.-Å. Larson and J. Zhou, "View matching for outer-join views," *The VLDB Journal*, vol. 16, no. 1, pp. 29–53, 2007.

[264] J. Lechtenbörger and G. Vossen, "On the computation of relational view complements," in *Proceedings of the 2002 ACM Symposium on Principles of Database Systems*, pp. 142–149, Madison, Wisconsin, USA, June 2002.

[265] J. Lechtenbörger and G. Vossen, "On the computation of relational view complements," *ACM Transactions on Database Systems*, vol. 28, no. 2, pp. 175–208, 2003.

[266] M. Lee and J. Hammer, "Speeding up materialized view selection in data warehouses using a randomized algorithm," *International Journal of Cooperative Information Systems*, vol. 10, no. 3, pp. 327–353, 2001.

[267] W. Lehner, "Query processing in data warehouses," in Liu and Özsu [291], pp. 2297–2301.

[268] W. Lehner, R. Cochrane, H. Pirahesh, and M. Zaharioudakis, "fAST refresh using mass query optimization," in *Proceedings of the 2001 International Conference on Data Engineering*, pp. 391–398, Heidelberg, Germany, April 2001.

[269] W. Lehner, R. Sidle, H. Pirahesh, and R. Cochrane, "Maintenance of automatic summary tables," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pp. 512–513, Dallas, Texas, USA, May 2000.

[270] M. Lenzerini, "Data integration: A theoretical perspective," in *Proceedings of the 2002 ACM Symposium on Principles of Database Systems*, pp. 233–246, Madison, Wisconsin, USA, June 2002.

[271] M. Lenzerini, "Data integration: A theoretical perspective," in *Proceedings of the 2002 ACM Symposium on Principles of Database Systems*, pp. 233–246, Madison, Wisconsin, USA, June 2002.

[272] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava, "Answering queries using views," in *Proceedings of the 1995 ACM Symposium on Principles of Database Systems*, pp. 95–104, San Jose, California, USA, June 1995.

[273] A. Y. Levy and Y. Sagiv, "Queries independent of updates," in *Proceedings of the 1993 International Conference on Very Large Data Bases*, pp. 171–181, Dublin, Ireland, August 1993.

[274] C. Li, "Rewriting queries using views," in Liu and Özsu [291], pp. 2438–2441.

[275] C. Li, M. Bawa, and J. D. Ullman, "Minimizing view sets without losing query-answering power," in *Proceedings of the 2001 International Conference on Database Theory*, pp. 99–113, London, UK, January 2001.

[276] J. Li, Z. Asgharzadeh Talebi, R. Chirkova, and Y. Fathi, "A formal model for the problem of view selection for aggregate queries," in *Proceedings of the 2005 East European Conference on Advances in Databases and Information Systems*, pp. 125–138, Tallinn, Estonia, September 2005.

[277] W.-S. Li, D. C. Zilio, V. S. Batra, M. Subramanian, C. Zuzarte, and I. Narang, "Load balancing for multi-tiered database systems through autonomic placement of materialized views," in *Proceedings of the 2006 International Conference on Data Engineering*, p. 102, Atlanta, Georgia, USA, April 2006.

[278] W. Liang, H. Li, H. Wang, and M. E. Orlowska, "Making multiple views self-maintainable in a data warehouse," *Data and Knowledge Engineering*, vol. 30, no. 2, pp. 121–134, 1999.

[279] W. Liang, H. Wang, and M. Orlowska, "Materialized view selection under the maintenance time constraint," *Data and Knowledge Engineering*, vol. 37, pp. 203–216, 2001.

[280] S. Lifschitz and M. A. V. Salles, "Autonomic index management," in *Proceedings of the 2005 International Conference on Autonomic Computing*, pp. 304–305, Seattle, Washington, USA, June 2005.

[281] S. Lightstone, "Seven software engineering principles for autonomic computing development," *Innovations in Systems and Software Engineering*, vol. 3, no. 1, pp. 71–74, 2007.

[282] S. Lightstone, G. M. Lohman, P. J. Haas, V. Markl, J. Rao, A. J. Storm, M. Surendra, and D. C. Zilio, "Making DB2 products self-managing: Strategies and experiences," *IEEE Data Engineering Bulletin*, vol. 29, no. 3, pp. 16–23, 2006.

[283] S. Lightstone, G. M. Lohman, and D. C. Zilio, "Toward autonomic computing with DB2 universal database," *ACM SIGMOD Record*, vol. 31, no. 3, pp. 55–61, 2002.

[284] S. Lightstone, M. Surendra, Y. Diao, S. S. Parekh, J. L. Hellerstein, K. Rose, A. J. Storm, and C. Garcia-Arellano, "Control theory: A foundational technique for self managing databases," in *ICDE Workshops*, pp. 395–403, 2007.

[285] S. Lightstone, T. Teorey, and T. Nadeau, *Physical Database Design: The Database Professional's Guide to Exploiting Indexes, Views, Storage, and More*. Morgan Kaufmann, 2007.

[286] B. G. Lindsay, L. M. Haas, C. Mohan, H. Pirahesh, and P. F. Wilms, "A snapshot differential refresh algorithm," in *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pp. 53–60, Washington DC, USA, May 1986.

[287] B. Liu, S. Chen, and E. A. Rundensteiner, "Batch data warehouse maintenance in dynamic environments," in *Proceedings of the 2002 International Conference on Information and Knowledge Management*, pp. 68–75, McLean, Virginia, USA, Novermber 2002.

[288] B. Liu, S. Chen, and E. A. Rundensteiner, "A transactional approach to parallel data warehouse maintenance," in *Proceedings of the 2002 International Conference on Data Warehousing and Knowledge Discovery*, pp. 307–316, Aix-en-Provence, France, September 2002.

[289] B. Liu and E. A. Rundensteiner, "Cost-driven general join view maintenance over distributed data sources," in *Proceedings of the 2005 International Conference on Data Engineering*, pp. 578–579, Tokyo, Japan, April 2005.

[290] B. Liu, E. A. Rundensteiner, and D. Finkel, "Restructuring batch view maintenance efficiently," in *Proceedings of the 2004 International Conference on*

Information and Knowledge Management, pp. 228–229, Washington DC, USA, Novermber 2004.

[291] L. Liu and M. T. Özsu, eds., *Encyclopedia of Database Systems*. Springer, 2009.

[292] Z. Liu and Y. Chen, "Answering keyword queries on XML using materialized views," in *Proceedings of the 2008 International Conference on Data Engineering*, pp. 1501–1503, Cancun, Mexico, April 2008.

[293] G. M. Lohman and S. Lightstone, "SMART: Making DB2 (more) autonomic," in *Proceedings of the 2002 International Conference on Very Large Data Bases*, pp. 877–879, Hong Kong, China, September 2002.

[294] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica, "Declarative networking," *Communications of the ACM*, vol. 52, no. 11, pp. 87–95, 2009.

[295] G. Luo, "V locking protocol for materialized aggregate join views on B-tree indices," in *Proceedings of the 2010 International Conference on Web-Age Information Management*, vol. 6184 of Lecture Notes in Computer Science, (L. Chen, C. Tang, J. Yang, and Y. Gao, eds.), pp. 768–780, Jiuzhaigou, Sichuan, China: Springer, July 2010. ISBN 978-3-642-14245-1.

[296] G. Luo, "Partial materialized views," in *Proceedings of the 2007 International Conference on Data Engineering*, pp. 756–765, Istanbul, Turkey, April 2007.

[297] G. Luo, J. F. Naughton, C. J. Ellmann, and M. Watzke, "A comparison of three methods for join view maintenance in parallel RDBMS," in *Proceedings of the 2003 International Conference on Data Engineering*, pp. 177–188, Bangalore, India, March 2003.

[298] G. Luo, J. F. Naughton, C. J. Ellmann, and M. Watzke, "Locking protocols for materialized aggregate join views," in *Proceedings of the 2003 International Conference on Very Large Data Bases*, pp. 596–607, Berlin, Germany, September 2003.

[299] G. Luo, J. F. Naughton, C. J. Ellmann, and M. Watzke, "Locking protocols for materialized aggregate join views," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 6, pp. 796–807, 2005.

[300] G. Luo and P. S. Yu, "Content-based filtering for efficient online materialized view maintenance," in *Proceedings of the 2008 International Conference on Information and Knowledge Management*, pp. 163–172, Napa Valley, California, USA, October 2008.

[301] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman, "Continuously adaptive continuous queries over streams," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, USA, June 2002.

[302] B. Mandhani and D. Suciu, "Query caching and view selection for XML databases," in *Proceedings of the 2005 International Conference on Very Large Data Bases*, pp. 469–480, Trondheim, Norway, August 2005.

[303] M. Marx, "Queries determined by views: Pack your views," in *Proceedings of the 2007 ACM Symposium on Principles of Database Systems*, pp. 23–30, Beijing, China, June 2007.

[304] G. Mecca, A. O. Mendelzon, and P. Merialdo, "Efficient queries over web views," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 6, pp. 1280–1298, 2002.

[305] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham, "Materialized view selection and maintenance using multi-query optimization," in *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pp. 307–318, Santa Barbara, California, USA, June 2001.

[306] M. K. Mohania and Y. Kambayashi, "Making aggregate views self-maintainable," *Data and Knowledge Engineering*, vol. 32, no. 1, pp. 87–109, 2000.

[307] I. S. Mumick, D. Quass, and B. S. Mumick, "Maintenance of data cubes and summary tables in a warehouse," in *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pp. 100–111, Tucson, Arizona, USA, May 1997.

[308] K. Munagala, J. Yang, and H. Yu, "Online view maintenance under a response-time constraint," in *Proceedings of the 2005 European Symposium on Algorithms*, pp. 677–688, Palma de Mallorca, Spain, October 2005.

[309] S. Muthukrishnan, "Data streams: Algorithms and applications," *Theoretical Computer Science*, vol. 1, 2006.

[310] A. Nash, L. Segoufin, and V. Vianu, "Views and queries: Determinacy and rewriting," *ACM Transactions on Database Systems*, vol. 35, no. 3, 2010.

[311] A. Nica, A. J. Lee, and E. A. Rundensteiner, "The CVS algorithm for view synchronization in evolvable large-scale information systems," in *Proceedings of the 1998 International Conference on Extending Database Technology*, pp. 359–373, Valencia, Spain, March 1998.

[312] N. F. Noy, A. Doan, and A. Y. Halevy, "Semantic integration," *AI Magazine*, vol. 26, no. 1, pp. 7–10, 2005.

[313] K. O'Gorman, D. Agrawal, and A. E. Abbadi, "Posse: A framework for optimizing incremental view maintenance at data warehouse," in *Proceedings of the 1999 International Conference on Data Warehousing and Knowledge Discovery*, pp. 106–115, Florence, Italy, September 1999.

[314] K. O'Gorman, D. Agrawal, and A. E. Abbadi, "On the importance of tuning in incremental view maintenance: An experience case study," in *Proceedings of the 2000 International Conference on Data Warehousing and Knowledge Discovery*, pp. 77–82, London, UK, September 2000.

[315] F. Olken and D. Rotem, "Random sampling from database files: A survey," in *Proceedings of the 1990 International Conference on Scientific and Statistical Database Management*, pp. 92–111, Charlotte, North Carolina, USA, April 1990.

[316] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. N. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, and X. Wang, "Nova: Continuous Pig/Hadoop workflows," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pp. 1081–1090, Athens, Greece, June 2011.

[317] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: A not-so-foreign language for data processing," in *Proceedings of the 2008 ACM*

SIGMOD International Conference on Management of Data, pp. 1099–1110, Vancouver, Canada, June 2008.

[318] P. E. O'Neil and D. Quass, "Improved query performance with variant indexes," in *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pp. 38–49, Tucson, Arizona, USA, May 1997.

[319] N. Onose, A. Deutsch, Y. Papakonstantinou, and E. Curtmola, "Rewriting nested XML queries using nested views," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pp. 443–454, Chicago, Illinois, USA, June 2006.

[320] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh, "Incremental maintenance for non-distributive aggregate functions," in *Proceedings of the 2002 International Conference on Very Large Data Bases*, pp. 802–813, Hong Kong, China, September 2002.

[321] S. Papadomanolakis and A. Ailamaki, "An integer linear programming approach to database design," in *ICDE Workshops*, pp. 442–449, 2007.

[322] S. Paraboschi, G. Sindoni, E. Baralis, and E. Teniente, "Materialized views in multidimensional databases," in *Multidimensional Databases*, pp. 222–251, Idea Group, 2003.

[323] C.-S. Park, M. Kim, and Y.-J. Lee, "Rewriting OLAP queries using materialized views and dimension hierarchies in data warehouses," in *Proceedings of the 2001 International Conference on Data Engineering*, pp. 515–523, Heidelberg, Germany, April 2001.

[324] C.-S. Park, M. Kim, and Y.-J. Lee, "Finding an efficient rewriting of OLAP queries using materialized views in data warehouses," *Decision Support Systems*, vol. 32, no. 4, pp. 379–399, 2002.

[325] N. W. Paton and O. Díaz, "Active database systems," *ACM Computing Surveys*, vol. 31, no. 1, pp. 63–103, 1999.

[326] T. Phan and W.-S. Li, "Dynamic materialization of query views for data warehouse workloads," in *Proceedings of the 2008 International Conference on Data Engineering*, pp. 436–445, Cancun, Mexico, April 2008.

[327] B. C. Pierce, "Linguistic foundations for bidirectional transformations: Invited tutorial," in *Proceedings of the 2012 ACM Symposium on Principles of Database Systems*, pp. 61–64, Scottsdale, Arizona, USA, May 2012.

[328] V. Poe, *Building a Data Warehouse for Decision Support*. Prentice Hall, 1996.

[329] A. Pol, C. M. Jermaine, and S. Arumugam, "Maintaining very large random samples using the geometric file," *The VLDB Journal*, vol. 17, no. 5, pp. 997–1018, 2008.

[330] L. Popa, M. Budiu, Y. Yu, and M. Isard, "DryadInc: Reusing work in large-scale computations," in *Proceedings of the 2009 Workshop on Hot Topics on Cloud Computing*, Boston, Massachusetts, USA, June 2009.

[331] X. Qian and G. Wiederhold, "Incremental recomputation of active relational expressions," *IEEE Transactions on Knowledge and Data Engineering*, vol. 3, no. 3, pp. 337–341, 1991.

[332] D. Quass, "Maintenance expressions for views with aggregation," in *Proceedings of the 1996 Workshop on Materialized Views*, pp. 110–118, 1996.

[333] D. Quass, A. Gupta, I. S. Mumick, and J. Widom, "Making views self-maintainable for data warehousing," in *Proceedings of the 1996 International*

Conference on Parallel and Distributed Information Systems, pp. 158–169, Miami Beach, Florida, USA, December 1996.

[334] D. Quass and J. Widom, "On-line warehouse view maintenance," in *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pp. 393–404, Tucson, Arizona, USA, May 1997.

[335] R. Ramakrishnan and J. Gehrke, *Database Management Systems*. McGraw-Hill, 3rd ed., 2009.

[336] C. Re and D. Suciu, "Materialized views in probabilistic databases for information exchange and query optimization," in *Proceedings of the 2007 International Conference on Very Large Data Bases*, pp. 51–62, Vienna, Austria, September 2007.

[337] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy, "Extending query rewriting techniques for fine-grained access control," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pp. 551–562, Paris, France, June 2004.

[338] K. A. Ross, "View adaptation," in Liu and Özsu [291], pp. 3324–3325.

[339] K. A. Ross, D. Srivastava, and S. Sudarshan, "Materialized view maintenance and integrity constraint checking: Trading space for time," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pp. 447–458, Montreal, Quebec, Canada, June 1996.

[340] N. Roussopoulos, "An incremental access method for ViewCache: Concept, algorithms, and cost analysis," *ACM Transactions on Database Systems*, vol. 16, no. 3, pp. 535–563, 1991.

[341] N. Roussopoulos, C.-M. Chen, S. Kelley, A. Delis, and Y. Papakonstantinou, "The ADMS project: Views "R" us," *IEEE Data Engineering Bulletin*, vol. 18, no. 2, pp. 19–28, 1995.

[342] N. Roussopoulos and H. Kang, "Principles and techniques in the design of ADMS±," *IEEE Computer*, vol. 19, no. 12, pp. 19–25, 1986.

[343] S. Rozen and D. Shasha, "A framework for automating physical database design," in *Proceedings of the 1991 International Conference on Very Large Data Bases*, pp. 401–411, Barcelona, Catalonia, Spain, 1991.

[344] G. Ruberg and M. Mattoso, "XCraft: Boosting the performance of active XML materialization," in *Proceedings of the 2008 International Conference on Extending Database Technology*, pp. 299–310, Nantes, France, March 2008.

[345] Y. Sagiv and M. Yannakakis, "Equivalences among relational expressions with the union and difference operators," *Journal of the ACM*, vol. 27, no. 4, pp. 633–655, 1980.

[346] K. Salem, K. S. Beyer, R. Cochrane, and B. G. Lindsay, "How to roll a join: Asynchronous incremental view maintenance," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pp. 129–140, Dallas, Texas, USA, May 2000.

[347] S. Samtani, V. Kumar, and M. K. Mohania, "Self maintenance of multiple views in data warehousing," in *Proceedings of the 1999 International Conference on Information and Knowledge Management*, pp. 292–299, Kansas City, Missouri, USA, Novermber 1999.

[348] A. D. Sarma, M. Theobald, and J. Widom, "LIVE: A lineage-supported versioned DBMS," in *Proceedings of the 2010 International Conference on Scientific and Statistical Database Management*, pp. 416–433, Heidelberg, Germany, June 2010.

[349] A. Sawires, J. Tatemura, O. Po, D. Agrawal, A. E. Abbadi, and K. S. Candan, "Maintaining XPath views in loosely coupled systems," in *Proceedings of the 2006 International Conference on Very Large Data Bases*, pp. 583–594, Seoul, Korea, September 2006.

[350] A. Sawires, J. Tatemura, O. Po, D. Agrawal, and K. S. Candan, "Incremental maintenance of path expression views," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pp. 443–454, Baltimore, Maryland, USA, June 2005.

[351] A. Segev and W. Fang, "Currency-based updates to distributed materialized views," in *Proceedings of the 1990 International Conference on Data Engineering*, pp. 512–520, Los Angeles, California, USA, February 1990.

[352] A. Segev and J. Park, "Maintaining materialized views in distributed databases," in *Proceedings of the 1989 International Conference on Data Engineering*, pp. 262–270, Los Angeles, California, USA, February 1989.

[353] A. Segev and J. Park, "Updating distributed materialized views," *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 2, pp. 173–184, 1989.

[354] L. Segoufin and V. Vianu, "Views and queries: Determinacy and rewriting," in *Proceedings of the 2005 ACM Symposium on Principles of Database Systems*, pp. 49–60, Baltimore, Maryland, USA, June 2005.

[355] P. Seshadri and A. N. Swami, "Generalized partial indexes," in *Proceedings of the 1995 International Conference on Data Engineering*, pp. 420–427, Taipei, Taiwan, March 1995.

[356] D. Shasha, "Tuning database design for high performance," in *The Computer Science and Engineering Handbook*, pp. 995–1011, CRC Press, 1997.

[357] D. Shasha, P. Bonnet, and J. Gray, *Database Tuning: Principles, Experiments and Troubleshooting Techniques*. Morgan Kaufmann, 2003.

[358] A. Shukla, P. Deshpande, and J. F. Naughton, "Materialized view selection for multidimensional datasets," in *Proceedings of the 1998 International Conference on Very Large Data Bases*, pp. 488–499, New York City, New York, USA, August 1998.

[359] A. Shukla, P. Deshpande, and J. F. Naughton, "Materialized view selection for multi-cube data models," in *Proceedings of the 2000 International Conference on Extending Database Technology*, pp. 269–284, Konstanz, Germany, March 2000.

[360] A. Simitsis and D. Theodoratos, "Data warehouse back-end tools," in *Encyclopedia of Data Warehousing and Mining*, (J. Wang, ed.), pp. 572–579, IGI Global, 2009.

[361] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis, "Dwarf: Shrinking the PetaCube," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pp. 464–475, Madison, Wisconsin, USA, June 2002.

[362] Y. Sismanis and N. Roussopoulos, "The complexity of fully materialized coalesced cubes," in *Proceedings of the 2004 International Conference on Very Large Data Bases*, pp. 540–551, Toronto, Canada, August 2004.

[363] D. Srivastava, S. Dar, H. V. Jagadish, and A. Y. Levy, "Answering queries with aggregation using views," in *Proceedings of the 1996 International Conference on Very Large Data Bases*, pp. 318–329, Mumbai (Bombay), India, September 1996.

[364] J. Srivastava and D. Rotem, "Analytical modeling of materialized view maintenance," in *Proceedings of the 1988 ACM Symposium on Principles of Database Systems*, pp. 126–134, Austin, Texas, USA, March 1988.

[365] M. Stonebraker, "The case for partial indexes," *ACM SIGMOD Record*, vol. 18, no. 4, pp. 4–11, 1989.

[366] K. Tajima and Y. Fukui, "Answering XPath queries over networks by sending minimal views," in *Proceedings of the 2004 International Conference on Very Large Data Bases*, pp. 48–59, Toronto, Canada, August 2004.

[367] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*. Addison-Wesley, 2005.

[368] W. C. Tan, "Provenance in databases: Past, current, and future," *IEEE Data Engineering Bulletin*, vol. 30, no. 4, pp. 3–12, 2007.

[369] N. Tang, J. X. Yu, M. T. Özsu, B. Choi, and K.-F. Wong, "Multiple materialized view selection for XPath query rewriting," in *Proceedings of the 2008 International Conference on Data Engineering*, pp. 873–882, Cancun, Mexico, April 2008.

[370] V. Tannen, "Relational algebra," in Liu and Özsu [291], pp. 2369–2370.

[371] I. Tatarinov, Z. G. Ives, J. Madhavan, A. Y. Halevy, D. Suciu, N. N. Dalvi, X. Dong, Y. Kadiyska, G. Miklau, and P. Mork, "The Piazza peer data management project," *ACM SIGMOD Record*, vol. 32, no. 3, pp. 47–52, 2003.

[372] M. Teschke and A. Ulbrich, "Concurrent warehouse maintenance without compromising session consistency," in *Proceedings of the 1998 International Conference on Database and Expert Systems Applications*, pp. 776–785, Vienna, Austria, August 1998.

[373] D. Theodoratos, "Detecting redundant materialized views in data warehouse evolution," *Information Systems*, vol. 26, no. 5, pp. 363–381, 2001.

[374] D. Theodoratos and M. Bouzeghoub, "Data currency quality satisfaction in the design of a data warehouse," *International Journal of Cooperative Information Systems*, vol. 10, no. 3, pp. 299–326, 2001.

[375] D. Theodoratos, S. Ligoudistianos, and T. K. Sellis, "View selection for designing the global data warehouse," *Data and Knowledge Engineering*, vol. 39, no. 3, pp. 219–240, 2001.

[376] D. Theodoratos and T. Sellis, "Data warehouse configuration," in *Proceedings of the 1997 International Conference on Very Large Data Bases*, pp. 126–135, Athens, Greece, August 1997.

[377] D. Theodoratos and T. Sellis, "Designing data warehouses," *Data and Knowledge Engineering*, vol. 31, pp. 279–301, 1999.

[378] D. Theodoratos and T. K. Sellis, "Incremental design of a data warehouse," *Journal of Intelligent Information Systems*, vol. 15, no. 1, pp. 7–27, 2000.

[379] D. Theodoratos, W. Xu, and A. Simitsis, "Materialized view selection for data warehouse design," in *Encyclopedia of Data Warehousing and Mining*, (J. Wang, ed.), pp. 1182–1187, IGI Global, 2009.

[380] A. Thiem and K.-U. Sattler, "An integrated approach to performance monitoring for autonomous tuning," in *Proceedings of the 2009 International Conference on Data Engineering*, pp. 1671–1678, Shanghai, China, March 2009.

[381] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive — a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[382] F. W. Tompa and J. A. Blakeley, "Maintaining materialized views without accessing base data," *Information Systems*, vol. 13, no. 4, pp. 393–406, 1988.

[383] O. G. Tsatalos, M. H. Solomon, and Y. E. Ioannidis, "The gmap: A versatile tool for physical data independence," in *Proceedings of the 1994 International Conference on Very Large Data Bases*, pp. 367–378, Santiago de Chile, Chile, September 1994.

[384] O. G. Tsatalos, M. H. Solomon, and Y. E. Ioannidis, "The GMAP: A versatile tool for physical data independence," *The VLDB Journal*, vol. 5, no. 2, pp. 101–118, 1996.

[385] J. D. Ullman, "Information integration using logical views," *Theoretical Computer Science*, vol. 239, no. 2, pp. 189–210, 2000.

[386] G. Valentin, M. Zuliani, D. C. Zilio, G. M. Lohman, and A. Skelley, "DB2 advisor: An optimizer smart enough to recommend its own indexes," in *Proceedings of the 2000 International Conference on Data Engineering*, pp. 101–110, Los Angeles, California, USA, February 2000.

[387] R. van der Meyden, "The complexity of querying indefinite data about linearly ordered domains," in *Proceedings of the 1992 ACM Symposium on Principles of Database Systems*, pp. 331–345, San Diego, CA, USA, June 1992.

[388] V. Vassalos, "Answering queries using views," in Liu and Özsu [291], pp. 92–98.

[389] Y. Velegrakis, "Side-effect-free view updates," in Liu and Özsu [291], pp. 2639–2642.

[390] Y. Velegrakis, "Updates through views," in Liu and Özsu [291], pp. 3244–3247.

[391] S. D. Viglas, J. F. Naughton, and J. Burger, "Maximizing the output rate of multi-way join queries over streaming information sources," in *Proceedings of the 2003 International Conference on Very Large Data Bases*, pp. 285–296, Berlin, Germany, September 2003.

[392] D. Vista, "Optimizing incremental view maintenance expressions in relational databases," PhD thesis, University of Toronto, 1996.

[393] D. Vista, "Integration of incremental view maintenance into query optimizers," in *Proceedings of the 1998 International Conference on Extending Database Technology*, pp. 374–388, Valencia, Spain, March 1998.

[394] J. Wang, M. J. Maher, and R. W. Topor, "Rewriting unions of general conjunctive queries using views," in *Proceedings of the 2002 International Conference on Extending Database Technology*, pp. 52–69, Prague, Czech Republic, March 2002.

[395] W. Wang, H. Lu, J. Feng, and J. X. Yu, "Condensed cube: An efficient approach to reducing data cube size," in *Proceedings of the 2002 International Conference on Data Engineering*, pp. 155–165, San Jose, California, USA, February 2002.

[396] G. Weikum, A. Mönkeberg, C. Hasse, and P. Zabback, "Self-tuning database technology and information services: From wishful thinking to viable engineering," in *Proceedings of the 2002 International Conference on Very Large Data Bases*, pp. 20–31, Hong Kong, China, September 2002.

[397] C. M. Wyss and E. L. Robertson, "Relational languages for metadata integration," *ACM Transactions on Database Systems*, vol. 30, no. 2, pp. 624–660, 2005.

[398] J. Xie and J. Yang, "A survey of join processing in data streams," in Aggarwal [15], pp. 209–236.

[399] M. Xu and C. I. Ezeife, "Maintaining horizontally partitioned warehouse views," in *Proceedings of the 2000 International Conference on Data Warehousing and Knowledge Discovery*, pp. 126–133, London, UK, September 2000.

[400] W. Xu, "The framework of an XML semantic caching system," in *Proceedings of the 2005 International Workshop on the Web and Databases*, pp. 127–132, Baltimore, Maryland, USA, June 2005.

[401] W. Xu and Z. M. Özsoyoglu, "Rewriting XPath queries using materialized views," in *Proceedings of the 2005 International Conference on Very Large Data Bases*, pp. 121–132, Trondheim, Norway, August 2005.

[402] W. Xu, C. Zuzarte, D. Theodoratos, and W. Ma, "Preprocessing for fast refreshing materialized views in DB2," in *Proceedings of the 2006 International Conference on Data Warehousing and Knowledge Discovery*, pp. 55–64, Krakow, Poland, September 2006.

[403] J. Yang, K. Karlapalem, and Q. Li, "Algorithms for materialized view design in data warehousing environment," in *Proceedings of the 1997 International Conference on Very Large Data Bases*, pp. 136–145, Athens, Greece, August 1997.

[404] J. Yang and J. Widom, "Incremental computation and maintenance of temporal aggregates," in *Proceedings of the 2001 International Conference on Data Engineering*, pp. 51–60, Heidelberg, Germany, April 2001.

[405] J. Yang and J. Widom, "Incremental computation and maintenance of temporal aggregates," *The VLDB Journal*, vol. 12, no. 3, pp. 262–283, 2003.

[406] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen, "Efficient maintenance of materialized top-k views," in *Proceedings of the 2003 International Conference on Data Engineering*, pp. 189–200, Bangalore, India, March 2003.

[407] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata, "Answering complex SQL queries using automatic summary tables," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pp. 105–116, Dallas, Texas, USA, May 2000.

[408] C. Zhang, J. Yang, and K. Karlapalem, "Dynamic materialized view selection in data warehouse environment," *Informatica (Slovenia)*, vol. 27, no. 4, pp. 451–460, 2003.

[409] C. Zhang, X. Yao, and J. Yang, "An evolutionary approach to materialized views selection in a data warehouse environment," *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, vol. 31, no. 3, pp. 282–294, 2001.

[410] X. Zhang, K. Dimitrova, L. Wang, M. El-Sayed, B. Murphy, B. Pielech, M. Mulchandani, L. Ding, and E. A. Rundensteiner, "Rainbow: Multi-XQuery optimization using materialized XML views," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, p. 671, San Diego, California, USA, June 2003.

[411] X. Zhang, L. Ding, and E. A. Rundensteiner, "PVM: Parallel view maintenance under concurrent data updates of distributed sources," in *Proceedings of the 2001 International Conference on Data Warehousing and Knowledge Discovery*, pp. 230–239, Munich, Germany, September 2001.

[412] X. Zhang, L. Ding, and E. A. Rundensteiner, "Parallel multisource view maintenance," *The VLDB Journal*, vol. 13, no. 1, pp. 22–48, 2004.

[413] X. Zhang and E. A. Rundensteiner, "DyDa: Dynamic data warehouse maintenance in a fully concurrent environment," in *Proceedings of the 2000 International Conference on Data Warehousing and Knowledge Discovery*, pp. 94–103, London, UK, September 2000.

[414] Z. Zhang and A. O. Mendelzon, "Authorization views and conditional query containment," in *Proceedings of the 2005 International Conference on Database Theory*, pp. 259–273, Edinburgh, UK, January 2005.

[415] J. Zhou, P.-Å. Larson, and H. G. Elmongui, "Lazy maintenance of materialized views," in *Proceedings of the 2007 International Conference on Very Large Data Bases*, pp. 231–242, Vienna, Austria, September 2007.

[416] J. Zhou, P.-Å. Larson, J. C. Freytag, and W. Lehner, "Efficient exploitation of similar subexpressions for query processing," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pp. 533–544, Beijing, China, June 2007.

[417] J. Zhou, P.-Å. Larson, J. Goldstein, and L. Ding, "Dynamic materialized views," in *Proceedings of the 2007 International Conference on Data Engineering*, pp. 526–535, Istanbul, Turkey, April 2007.

[418] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom, "View maintenance in a warehousing environment," in *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pp. 316–327, San Jose, California, USA, May 1995.

[419] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener, "The strobe algorithms for multi-source warehouse consistency," in *Proceedings of the 1996 International Conference on Parallel and Distributed Information Systems*, pp. 146–157, Miami Beach, Florida, USA, December 1996.

[420] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener, "Multiple view consistency for data warehousing," in *Proceedings of the 1997 International Conference on Data Engineering*, pp. 289–300, Birmingham, UK, April 1997.

[421] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener, "Consistency algorithms for multi-source warehouse view maintenance," *Distributed and Parallel Databases*, vol. 6, no. 1, pp. 7–40, 1998.

[422] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. J. Storm, C. Garcia-Arellano, and S. Fadden, "DB2 Design Advisor: Integrated automatic physical database design," in *Proceedings of the 2004 International Conference on Very Large Data Bases*, pp. 1087–1097, Toronto, Canada, August 2004.

[423] D. C. Zilio, C. Zuzarte, S. Lightstone, W. Ma, G. M. Lohman, R. Cochrane, H. Pirahesh, L. S. Colby, J. Gryz, E. Alton, D. Liang, and G. Valentin, "Recommending views and indexes with IBM DB2 design advisor," in *Proceedings of the 2004 International Conference on Autonomic Computing*, pp. 180–188, New York City, New York, USA, May 2004.