Perturbation Analysis of Database Queries*

Brett Walenz
Duke University
bwalenz@cs.duke.edu

Jun Yang
Duke University
junyang@cs.duke.edu

ABSTRACT

We present a system, *Perada*, for parallel *perturbation analysis* of database queries. Perturbation analysis considers the results of a query evaluated with (a typically large number of) different parameter settings, to help discover leads and evaluate claims from data. Perada simplifies the development of general, ad hoc perturbation analysis by providing a flexible API to support a variety of optimizations such as grouping, memoization, and pruning; by automatically optimizing performance through run-time observation, learning, and adaptation; and by hiding the complexity of concurrency and failures from its developers. We demonstrate Perada's efficacy and efficiency with real workloads applying perturbation analysis to computational journalism.

1 Introduction

Data-driven decision making is playing an increasingly important role today in many domains, from health, business, sports and entertainment, to public policy. Decisions are often driven by database queries, which derive insights, identify trends and issues, and justify actions. Asking the right queries is critical to decision making. "Bad" queries (intentionally or otherwise) can present cherry-picked views, introduce bias, and result in misleading or potentially dangerous conclusions. By "tweaking" a database query in various ways and studying how its result changes, *perturbation analysis* provides a framework for finding the right queries and guarding against bad ones.

A noteworthy application of perturbation analysis is in *computational journalism* [7, 6], and particularly in finding leads and checking facts using data.

Example 1 (CBB/streak)**.** *"Danny Ferry scored* 11 *points or more in* 33 *consecutive games during the 1988–89 season. Only once has this record been beaten in the history of Duke University Men's Basketball."*

Sports reporters and team publicists routinely come up with this kind of claim (called prominent streaks *in [24]). Naturally, one wonders how we can find such claims, and whether they are truly significant. Perturbation analysis offers us insights to both questions. By considering other players and different runs of games, we can obtain all combinations of point minimum and streak length, and further select those dominated by at most one other instance. The final set of results gives us all claims that are as "interesting" as the original one (thereby finding leads). The size of the final set also tells us how unique the original claim is (thereby checking facts)—if analogous claims can be made about many other players, then Danny Ferry's streak is not really significant, even though it appears so (being dominated by only one other instance).*

Example 2 (Congress/vote, adapted from [22])**.** *"Jim Marshall, a Democratic incumbent from Georgia, voted the same as Republican Leader John Boehner 65 percent of the time in 2010." This claim came from a TV ad in the 2010 elections and was checked by* factcheck.org.

Campaign ads frequently invoke such claims to strengthen their own candidates or to attack opponents. Again, perturbation analysis can find and check such claims. By perturbing the legislators involved in comparison, we can see what other pairs of legislators from different parties have similarly high agreement percentages, and evaluate how unique the original claim is in the broader context. Moreover, by perturbing the time period of comparison, we see how robust the claim is. Here, if we expand the comparison to 2007–2010, then Marshall agreed with Boehner only 56 percent of the time, which is not very high compared with other Democrats. These and other uses of perturbation analysis for fact-checking are discussed in depth in [22].

Conceptually, both examples follow a similar procedure of perturbation analysis: given a database and a parameterized query "template," we evaluate the query with various parameter settings to obtain a set of results (or "perturbations") representing the "bigger picture"; then we filter, rank, or summarize the results to find interesting claims or to assess the significance of particular claims. Even if the database itself is not big, the space of perturbations can be enormous. With current technologies, timely perturbation analysis remains challenging.

Using a database system, a developer familiar with SQL can write the query template as a prepared SQL statement, and evaluate it with each parameter setting; the results can be collected into a table and then post-processed using another SQL query. An experienced SQL developer may even be able to write the entire analysis as a single SQL query—albeit a very complex one, with the help of view definitions and/or `WITH` clauses. However, both approaches are prohibitively expensive (as we will experimentally verify later) as database systems are not designed for this type of workloads.

*The authors are supported by NSF grants IIS-1408846 and IIS-1320357, a Google Faculty Research Award, and Google Research Cloud Credits. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the funding agencies.

Proceedings of the VLDB Endowment, Vol. 9, No. 14

Another approach is to develop specialized algorithms for specific query templates and applications, as in, e.g., [21, 24, 22]. Unfortunately, developing such algorithms for ad hoc situations requires a level of expertise and effort that is simply unavailable to typical users. Data journalism, for example, has been called "social science done on deadline," [16] and is often performed with very constrained resources. Because of the time, expertise, and cost required by their development, efficient specialized algorithms often turn out to be less practical than easy-to-implement solutions that are efficient "just enough" to get the job done.

Therefore, there is a need for a better system for perturbation analysis. This system needs to be easy to scale to a large number of perturbations under a deadline. It needs to be general, with versatile optimization techniques applicable to different query templates. It needs to be simple to use—developers should be able to program ad hoc analysis quickly, without worrying about low-level implementation details and performance tuning.

We are developing a system called ***Perada*** (***Per****turbation* ***a****nalysis of* ***da****tabase queries*) to meet these requirements. A key challenge is the degree to which we can automate optimization while still delivering acceptable performance for complex ad hoc perturbation analysis. Perada identifies a suite of optimizations that are generally effective in perturbation analysis: *parallelization* (trivially, we can evaluate all perturbations in parallel); *grouping* (we can evaluate some perturbations more efficiently as a group); *memoization* (we can cache the result of a piece of computation and reuse it when we encounter the same inputs later); *pruning* and *incremental computation* (we can use the results cached for one set of inputs in more sophisticated ways to help prune or compute for a different set of inputs). Perada provides hooks for developers to enable these optimizations easily.

More importantly, developers are not committing themselves to invoking all these optimizations in particular ways. As we will see later, performance depends on appropriate optimization settings—e.g., whether memoization and pruning are worth their overhead, or how to strike a balance between parallelism and serialism (the latter of which enables memoization, pruning, and incremental computation). However, appropriate settings may depend on factors difficult to foresee by developers, such as data characteristics. Perada removes the burden of specifying optimization settings from developers; instead, it monitors execution and adjusts optimization settings dynamically and adaptively.

Therefore, with Perada, developers can focus on specifying the analysis and optimization opportunities through the Perada API (using Python and SQL); Perada implements useful functions such as distributed caching and parallel randomized enumeration of parameter settings, and handles all details of tuning and execution using modern cluster computing tools (Spark [10] and Redis [13]). Perada also protects developers from some rather nasty implementation pitfalls that arise from failures and concurrency. In this paper, we describe the design and implementation of Perada, show how it helps us implement perturbation analysis for a number of real-life scenarios in computational journalism, and demonstrate its efficacy and efficiency through experiments.

2 Problem Statement

Suppose we have a database $\mathcal{D}$. We are given a parameterized *query template* q over $\mathcal{D}$ with parameters settings drawn from a (typically multi-dimensional) *parameter space* P. Let $q(p)$ denote the result of evaluating q over $\mathcal{D}$ with parameter setting $p \in P$. Let $\mathcal{R} = \{(p, q(p)) \mid p \in P\}$ denote the collection of results (or *perturbations*) obtained by evaluating q over the entire parameter space. We are further given a *post-processing query* χ over $\mathcal{R}$, which computes the final *answer set* for the perturbation analysis. Hence, a perturbation analysis problem is specified by $(\mathcal{D}, P, q, \chi)$.

The post-processing query χ may simply filter or aggregate $\mathcal{R}$. However, in many cases, χ can do more, such as ranking or clustering $\mathcal{R}$, so in general, it may be impossible to determine whether or how much a particular perturbation contributes to the final answer set without examining other perturbations. The following examples illustrate a range of possibilities for the problem definition. For ease of exposition, we have simplified the database schemas below (some tables are in reality join views).

Example 3 (CBB/streak)**.** *Recall Example 1. Here:*

- *The database $\mathcal{D}$ records per-player and per-game statistics for Duke Men's Basketball Team in a table* (season, game-date, player, points, rebounds, . . .).
- *The query template q is parameterized by* (player, start-date, end-date)*, where the dates specify a sequence of consecutive games in the same season played by* player*. The query computes* (length, points)*, where* length *is the number of games between* start-date *and* end-date*, and* points *is the minimum* points *scored by the given* player *in these games.*
- *Given the set of perturbations $\mathcal{R}$, the post-processing query χ returns those in $\mathcal{R}$ whose* (length, points) *pairs are "dominated" by no more than κ other pairs ($\kappa = 1$ in Example 1). We say that (x, y) dominates (x', y') if $x \geq x'$ and $y \geq y'$ and at least one inequality holds.*

Example 4 (Congress/vote)**.** *Recall Example 2. Here:*

- *$\mathcal{D}$ (source:* govtrack.us*) records the votes by each legislator in each roll call of the U.S. Congress in a table* (person, party, roll-call-id, date, vote).
- *q is parameterized by* (person1, person2, start-year-month, end-year-month)*, where* person1 *is a Democratic legislator and* person2 *is a Republican legislator. The query computes a single number* agree%*, the percentage of the times that the two legislators agree in their votes during the specified time period.*
- *χ simply selects those perturbations with* agree% $\geq \tau$ *(in Example 2, $\tau = 65$).*

Example 5 (MLB/dom)**.** *"Only one other player has ever beaten Ryan Howard's 2006 season where he had* 182 *hits,* 58 *home runs, and a* .310 *batting average." This type of claims are called* "one of the few" *in [21]. Here:*

- *$\mathcal{D}$ records statistics for Major League Baseball players by season and by player, in a table* (season, player, hits, home-runs, batting-avg, earned-run-avg, . . .)*. More than twenty stat columns are available.*
- *q is parameterized by* (player, season)*, and computes* domcount*, the number of rows in the stats table that dominate the row with the given* (player, season) *on a set of stat columns $\mathcal{M}$ (in the claim above, $\mathcal{M} =$* {hits, home-runs, batting-avg}*). The notion of dominance here is a straightforward generalization of that in Example 3 to higher dimensions; also, for some stats smaller values are better (e.g.,* hits-allowed *for pitchers).*
- *χ simply selects those perturbations with* domcount $\geq \kappa$ *(in the claim above, $\kappa = 1$).*

This example can be seen as a simpler version of Example 3. Here, the measures involved in dominance test are already stored in the database; in Example 3, they have to be computed first, so dominance computation occurs in χ there.

3 Opportunities and Challenges

Beyond Parallelization The simplest way to speed up perturbation analysis is to parallelize. The bulk of the workload is indeed embarrassingly parallel: each perturbation can be evaluated independently. However, this approach misses several opportunities mentioned in Section 1. We present several examples of how grouping, memoization, and pruning help.

Example 6 (CBB/streak; grouping)**.** *Recall Example 3. Consider all perturbations pertaining to a given* player *and* season*. All valid* (start-date, end-date) *intervals correspond to different subsequences of games in the* season*, and there is considerable overlap among these subsequences. Rather than querying each subsequence independently, it is more efficient to compute all perturbations for the same* (player, season) *in one go.*

Example 7 (MLB/dom; memoization)**.** *Recall Example 5 and suppose we are interested in stats* $\mathcal{M} = \{\text{hits}, \text{home-runs}\}$*. Suppose* (x, y) *is the stats for some* (player, season)*. Once we compute* domcount *for* (x, y)*—with a* 2*-d range aggregation query against* $\mathcal{D}$*—we can remember this result and have it keyed by* (x, y)*. Later, if we encounter another* (player, season) *that happens to yield the same* (x, y)*, we can reuse the result remembered, avoiding a dominance counting query. How often does memoization help? Surprisingly often for some workloads. In this case, there are* 656,214 (player, season) *pairs, but they map to only* 8,791 *distinct* (hits, home-runs) *value pairs, because these two stats have relatively small value domains. Thus, memoization would result in a* 98% *reduction in the number of dominance counting queries.*

Example 8 (MLB/dom; pruning)**.** *Continuing with the above example, further suppose that* $\kappa = 10$*; i.e., we are interested in performances dominated no more than ten other occurrences. As in the above example, once we find that a stat value pair* (x, y) *has* domcount c*, we will remember* (x, y, c)*. If we encounter a previously unseen pair* (x', y')*, instead of immediately computing its* domcount *on* $\mathcal{D}$*, we first check those entries we remembered to see if we can find some entry* $(x^\star, y^\star, c^\star)$ *where* $(x^\star, y^\star)$ *dominates* (x', y') *and* $c^\star \geq 10$*. If yes, then we know, by transitivity of dominance, that the current perturbation we are considering has no chance of passing* χ *because its* domcount *is at least* 11*, even though we do not know exactly what it is. (This example can be easily expanded to illustrate incremental computation; see Appendix A.2 for details.)*

Note that Example 8 above illustrates how we can "push down" a filter in the post-processing query χ into the computation of perturbations, and then using this filter to help prune computation. Pruning based on χ can give substantial savings, but to expose such opportunities in general, we need to do more than simply pushing down predicates "local" to a perturbation. Shortly, at the beginning of Example 9, we shall see how pruning works in CBB/streak even though its χ is based on an aggregate over all perturbations.

Perada needs to provide mechanisms to exploit these opportunities in a way that allows them to work together effectively. First, we need to strike a balance between parallelism and sequentiality. To one extreme, evaluating each perturbation independently maximizes the degree of parallelism, but there would be no saving from memoization, pruning, and incremental computation because they rely on evaluations done earlier in time. To the other extreme, serial execution maximizes opportunities of reusing earlier evaluations, but we lose the speedup from parallelization. A similar trade-off exists for the granularity of parallelism, where grouping may favor coarser grains of parallelism. Our challenge is to design a system that allows such trade-offs to be explored, rather than hard-coded.

Caching Challenges With parallel execution in a cluster of nodes, the conceptually simplest method to enable memoization, pruning, and incremental computation is through a global cache, which allows information from earlier units of computation to be passed to later ones (even if they run on different nodes). Memoization, as illustrated by Example 7, can be easily realized using any one of high-performance distributed key-value store available today (we use Redis [13]). On the other hand, pruning and incremental computation generally requires a much richer interface for accessing cached data. For instance, pruning in Example 8 involves a multi-dimensional range query against the cache. Ideally, we would like the cache to be a SQL database, which offers proper indexing and querying support. However, challenges arise in coping with concurrency and failures, while keeping the overhead low to preserve the benefit of pruning. To illustrate, we use an example.

Example 9 (CBB/streak; pruning and caching challenges)**.** *Consider the following possible parallel implementation of CBB/streak (recall Example 3). We fire up multiple nodes in a cluster to find streaks for different* (player, season) *pairs. Meanwhile, we maintain a global SQL cache of* (length, points) *entries for streaks found so far, to help prune perturbations that cannot be in the final answer. Specifically, once we discover a streak with* length x *and* points y*, we check if* (x, y) *is dominated by two or more entries already in the cache (recall that* $\kappa = 1$*). If yes, we simply stop considering the current streak, because its* domcount *can only increase further as the execution progresses and we see more streaks. Otherwise, we continue, and insert* (x, y) *into the cache. This example illustrates how we can prune based on* χ *even though* χ *cannot be "pushed down" simply (as we cannot evaluate* domcount *here locally with a single perturbation), as alluded to earlier.*

Suppose, however, that the node fails right after inserting (x, y) *but before it completes processing the current streak. To recover, we need to redo this perturbation, which may cause* (x, y) *to be inserted again into the cache. Subsequently, streaks dominated by* (x, y) *can be incorrectly pruned because of the two copies of* (x, y)*, only one of which should be counted in reality.*

Workarounds are possible for this simple, specific example. For instance, we could make insertions into the cache idempotent, by tagging each insertion by its perturbation id, and ignoring an insertion if an entry with the same id already exists. However, this workaround fails if we want to find streaks that are dominated or matched *at most once (with the pruning query against the cache adjusted accordingly). Suppose* (x, y) *is dominated by one other streak but never matched. During redo, however, the pruning query will pick up both the dominating streak and the earlier inserted* (x, y)*, thereby rejecting* (x, y) *incorrectly. To fix the problem, we would also have to rewrite the pruning query to ignore cache entries with the same id as the perturbation being processed.*

But this fix may still not be general enough when developers modify a SQL cache in ways other than insertions. For instance, a developer may decide to keep the cache small by maintaining only the 2*-skyband [15] of the streaks seen so far (which is sufficient for pruning). In this case, seeing a new streak may cause deletions from the cache as well.*

As the example above shows, it is unrealistic to rely on developers to devise adequate workarounds to handle potential failures. One alternative is to rely on the standard transaction support of a database system. However, operating a database system as a global SQL cache with transactional updates incurs considerable overhead. Our challenge is to provide a highly efficient caching layer with good abstraction that frees developers from having to worry about concurrency and failures. From a performance perspective,

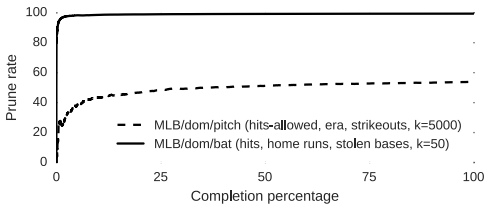

Figure 1: Pruning rate over the course of serially executing MLB/dom, with different κ and different stats for testing dominance.

the bottom line is that information exchange among different execution threads incurs cost, which can add up and offset the benefit of pruning. While immediate information exchange maximizes pruning opportunities, doing so with a global SQL cache may not pay off because of its overhead. We would like to offer the convenience of a SQL cache, while relaxing its transactional semantics to offer more flexible control over when to exchange information globally; we must also do so with clean semantics to avoid the potential pitfalls illustrated in Example 9.

Optimization Challenges As discussed earlier in this section, various trade-offs may affect the performance of perturbation analysis. We now argue that simply exposing these "knobs" to developers is not enough, as they can be difficult to tune by hand.

Example 10 (MLB/dom; factors affecting pruning rate)**.** *Consider MLB/dom, with the pruning procedure described in Example 8. Suppose* $\mathcal{M} = \{\text{hits}, \text{home-runs}, \text{stolen-bases}\}$ *and* $\kappa = 50$*. Figure 1 (ignore the dashed-line plot for now) shows the pruning rate (averaged over a moving window) over the entire course of execution. Near the beginning of execution, we see that the pruning rate picks up quickly, as we are accumulating a good sample of perturbations in the cache for pruning. Once we have a reasonable approximation of the "decision boundary" separating stat tuples inside and outside the final answer set—geometrically, the last tier of the* $(\kappa+1)$*-skyband in the* 3*-d space of stat points—the pruning rate plateaus. This effect of "diminishing returns" should inform our choice of when to exchange information among parallel execution threads: we should exchange more often early on and less over time, as the benefit of growing a global cache decreases.*

Now consider the same analysis, but with $\mathcal{M} = \{$hits-allowed, strikeouts, earned-run-average$\}$ *and* $\kappa = 5000$*. Figure 1 also plots the pruning rate over the course of executing this analysis. Comparing with the other case, we see that the pruning rate here increases more slowly, and converges to a lower percentage. The reason is with a bigger* κ *and the fact that* strikeouts *and* earned-run-average *tend to be anti-correlated, the decision boundary (last tier of the* 5001*-skyband in* 3*-d) is much bigger, and the fraction of perturbations that cannot be pruned (because they are in the answer set) is also bigger.*

As this example illustrates, the appropriate setting for the frequency of information exchange changes over the course of execution, and can differ for different applications of the same analysis, even on the same dataset. We also note that this example simplifies reality: it uses no parallelization or memoization, both of which can affect the pruning rate. In reality, these additional factors, as well as data characteristics, make tuning optimization knobs by hand impractical. Our challenge is to enable Perada to make intelligent optimization decisions dynamically and automatically.

4 System

Overview We now discuss how Perada addresses the challenges presented in Section 3. At a high level, Perada provides an API to program perturbation analysis as a sequence of *jobs*, where each

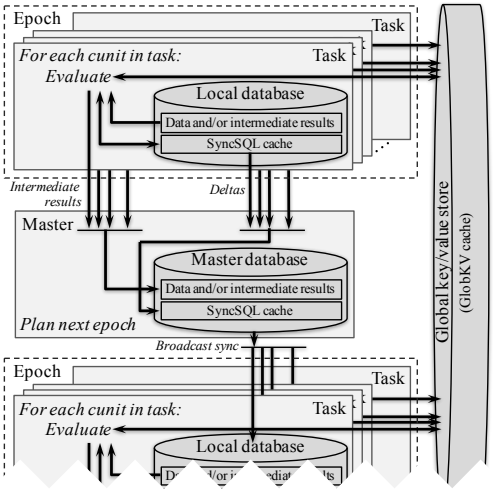

Figure 2: Overview of Perada job execution and optimization.

executes as a (usually large) number of *cunits* (*units of computation*) on a cluster, thereby enabling parallelization. Each cunit can evaluate one or more perturbations (with grouping optimization).

To enable exchange of information that helps memoization, pruning, and incremental computation, Perada provides two types of caches for cunits to use. The *GlobKV* cache is a global key/value store for supporting memoization; one cunit's results become immediately visible to others. The *SyncSQL* cache is a SQL database local to each thread of execution for supporting pruning and incremental computation. To keep overhead low, its contents are synchronized across the cluster only at specific times. Behind the scene, Perada controls the timing dynamically by dividing job execution adaptively into a sequence of *epochs*, with synchronization only on epoch boundaries.

Perada provides several other features to simplify development and tuning. It supports parallel random sampling of the parameter space by default, which makes many pruning methods effective. With the Perada API, developers can specify *ochoices*, Boolean flags controlling which specific optimizations in the code to enable. Instead of relying on developers to set ochoices, Perada explores them at run time to find the optimal combination to apply. Finally, Perada recovers from cluster node failures automatically.

Figure 2 illustrates how Perada executes a job in parallel using the two types of caches, and how it dynamically optimizes the remaining work at the end of each epoch. The remainder of this section describes the Perada system and API in detail; we leave the automatic optimizer to Section 5.

4.1 Parallel Execution

As mentioned earlier, a Perada program consists of a sequence of *jobs*, and each job consists of a number of *cunits*, each a unit of computation indivisible from Perada's perspective. All cunits in a job run the same code but work on different inputs, which we call *cunit parameters*. The input database $\mathcal{D}$ to the program is available read-only to all cunits in all jobs. The output of a job is the collection of output from all cunits in the job. The final answer set is the output of the last job in the program. We start with a simple, single-job example below:

Example 11 (MLB/dom; cunits)**.** *Recall Example 5. We implement this analysis with a single job, where each cunit is parameterized by a* (player, season) *pair. The cunit determines whether the given stat record's* domcount *with respect to* $\mathcal{M}$ *is at least* κ*; if yes, it outputs* (player, season, domcount)*. Here, there is no grouping, so each cunit evaluates one perturbation and applies the* χ *filter to it.*

Our next example uses multiple jobs and grouping. In a program with multiple jobs, the output of a job is accessible to all subsequent jobs, either as a read-only database, or as input cunit parameters; for the former, the Perada API allows an relational schema to be declared for the output. Note that cunit parameters in general are not necessarily from the parameter space P of the perturbation analysis—they may come from earlier job output, or, because of grouping, they may be "meta-parameters" each corresponding to a group of parameter settings in P.

Example 12 (CBB/streak; multiple jobs and grouping)**.** *Recall Example 3. We use two jobs to implement this analysis. In the first job, each cunit is parameterized by a* (player, season) *pair as described in Example 6, and is responsible for considering all perturbations with parameters* (player, start-date, end-date) *where the dates specify a sequence of consecutive games by* player *in* season*. Using caching (to be further discussed in Section 4.2), the cunit outputs a perturbation* (player, start-date, end-date, length, points) *only if it can potentially contribute to the final answer set (we cannot know for sure at this point because we have not seen all streaks yet).*

We make the output set of candidate streaks from the first job available as a database to the second job. The second job is similar to the job in Example 11. Each cunit considers one candidate streak, and outputs it with its domcount *in the candidate set (with respect to* (length, points)*) if* domcount $\leq \kappa$*.*

Note that both jobs contribute to the evaluation of χ: the first job prunes away some perturbations from further consideration, while the second job further processes the remaining candidates to find the final answer set.

As mentioned in the overview, we divide all cunits in a job into a sequence of *epochs*. Suppose we have n slots available in the cluster for parallel execution. Given this limit, Perada further partitions the cunits in an epoch in into nw *tasks*, where w, the *number of waves*, is a small integer. Within an epoch, tasks are scheduled to run on the n slots independently of each other. Within each task, cunits execute serially. We defer the discussion of how cunits share information through caching to Section 4.2.

Implementation with Spark Perada currently uses Spark [10] as the underlying platform for parallel execution. Developers are not directly exposed to the Spark API, so it is possible to replace Spark with other platforms should the need arise. Perada runs a *master* that manages a Spark cluster. A Perada job is a Spark job, a Perada epoch corresponds to a Spark stage, and a Perada task runs as a Spark task. However, our use of Spark stages departs from their standard usage. In Spark, stages are derived from a workflow graph and generally involve different computation. However, Perada epochs in the same job consist of identical computation on different inputs. As we will further discuss in Sections 4.2 and 5.3, we purposefully divide parallelizable computation into a sequence of epochs to control the rate of information exchange through the SyncSQL cache. By carefully constructing the Spark workflow, Perada ensures its desired mapping of epochs to Spark stages.

Using Spark's broadcast mechanism, Perada replicates the input database $\mathcal{D}$ as well as any database created from earlier job output. The data is made available as an in-memory read-only SQLite [1] database local to each task. If data size ever becomes an issue, it is easy to replace the replicated database with a distributed one without affecting user code. We have not found the need to do so for our workloads, because the scalability challenge for perturbation analysis tends to arise from the large space of perturbations instead of a large $\mathcal{D}$. In the event that intermediate job output is large, Perada offers the option of leaving output distributed in the Spark cluster instead of putting it into a single database.

Perada relies on Spark's built-in scheduler to assign the nw tasks in each epoch to the n slots. By setting $w > 1$, Spark can offer some automatic protection against stragglers. In our experience, however, we found $w = 1$ to suffice because by default we randomly assign a number of cunits to each task, so the chance for any task to receive a disproportionally high amount of work is low.

Perada also relies on Spark for failure recovery—a failed task will be rerun automatically. Because of caching, however, complications may arise as illustrated in Example 9. We discuss how Perada handles these complications in the next section.

4.2 Caching

Perada recognizes the different cache usage patterns in perturbation analysis and the different levels of performance offered by various caching architectures—there is no "one-size-fits-all." Hence, Perada offers two types of caches, which we describe in detail below.

***GlobKV*: Global Key/Value Cache** This cache is intended for memoization. It supports storage of key/value pairs, and equality lookups by keys. Importantly, Perada disallows updates that change values—it will raise an error if one attempts to store a different value with an existing key. Perada exposes this restriction to developers, to guard against possible misuses of GlobKV as a general global store, which could lead to subtle bugs involving concurrency and failures. This restriction does not limit memoization, because if we are caching the result of a function using its input as the key, there cannot be different results with the same key. The following example illustrates the use of the GlobKV cache.

Example 13 (MLB/dom; GlobKV)**.** *Recall Example 7. Cunits can use the GlobKV cache to store and look up stats-to-*domcount *mappings. As soon as a cunit adds an entry to the GlobKV cache, the entry will benefit all cunits that follow in time, be they from the same or other tasks.*

Note that two cunits from two concurrent tasks could examine the same combination of stat values at about the same time. It is possible that both cunits check the cache and fail to find this key, so both proceed with domcount *evaluation and write to the cache. However, both should compute the same* domcount *value, and the order of writes does not matter. In this case we did not avoid redundant computation, but such occurrences do not affect correctness, and should be quite rare.*

Perada implements the GlobKV cache using Redis [13], though any other scalable modern key/value store will do. Note that the GlobKV cache also helps with failure recovery: when we redo a failed task, its cunits will benefit from any entries that were computed and cached earlier by the failed run.

***SyncSQL*: Local SQL Caches with Between-Epoch Synchronization** This cache supports pruning and incremental computation by offering SQL access to cached data, which is more powerful than the GlobKV cache. The SyncSQL cache operates as follows. The Perada master maintains a *master cache*, whose contents are replicated to a database local to each task at the beginning of each epoch. The cunits in the task can interact with the local database with full SQL. No information is exchanged directly among the local databases. At the end of the task, Perada computes a *delta*—which represents the new information added to the local database by this task's cunits—and ships it to the master. The master consolidates the deltas from all tasks and updates the master cache at the end of the epoch. The new version of the master cache will then be replicated for the next epoch.

This design allows information to be passed to later epochs, but not between tasks within the same epoch. A cunit can benefit from the collective knowledge of all cunits in previous epochs, as well as

those cunits that precede this one in the same task; however, it does not benefit from cunits in other tasks of the same epoch. By adjusting the epoch sizes (discussed further in Section 5.3), Perada can explore the trade-off between the cost and benefit of information exchange among cunits.

The program needs to specify a function `gen_delta` for computing the delta from a local SyncSQL database, and a function `apply_deltas` for applying the deltas to the master cache. Perada provides default implementations: `gen_delta` by default returns all entries in a local database at the end of the task that were not present at the beginning of the task, and `app_deltas` simply inserts in all such entries into the master cache. Although developers need to supply these two functions or verify that their default implementations suffice, they have unfettered SQL access to a local database without having to worry about tricky issues involving concurrency and failures, such as those in Example 9.

The following example illustrates the use of the SyncSQL cache:

Example 14 (MLB/dom; SyncSQL)**.** *Recall Example 8. We show how to use the SyncSQL cache for pruning (in a more efficient way than Example 8). Suppose a cunit is considering a perturbation with stat values* (x, y)*. We query the local SyncSQL database:*

*SELECT MAX(*domcount *+*
(CASE WHEN hits *>* x *OR* home-runs *>* y *THEN 1 ELSE 0))*
FROM cache *WHERE* hits *>=* x *AND* home-runs *>=* y*;*

If the result is (non-NULL and) greater than κ*, we can prune the perturbation. Otherwise, we compute* domcount *for* (x, y) *on* $\mathcal{D}$*; suppose it is* c*. If* $c \geq \kappa$ *(which means* (x, y, c) *can be useful later for pruning), we update the local SyncSQL database as follows:*

DELETE FROM cache *WHERE* hits *<=* x *AND* home-runs *<=* y*;*
INSERT INTO cache *VALUES(*x*,* y*,* c*);*

Note that the DELETE statement does not decrease the pruning power of the cache, because any perturbation that can be pruned by a deleted entry can still be pruned by (x, y, c)*.*

The default `gen_delta` implementation works perfectly for this SyncSQL cache. For `app_delta`, we insert all delta entries into the master cache, and then delete all but the skyline entries (with respect to the stat columns of interest). Note that if we use the default `app_delta` implementation, pruning will still work, although we end up with larger master cache to replicate for the next epoch.

Perada implements each local SyncSQL database using SQLite. Each task calls `gen_delta` after completing all its cunits, and the Perada master collects all resulting deltas via Spark. The master then calls `apply_deltas` to update the master cache, and uses Spark's broadcast mechanism to replicate it for tasks in the next epoch. Currently, our default implementation of `gen_delta` is rather naive; a more efficient implementation should be possible by examining the SQLite log.

Note that if a task fails, its associated local SyncSQL database is simply lost; any SyncSQL cache updates performed by this failed task have no effect on the rest of execution, including when redoing the same task. The Perada master receives deltas only from successfully completed tasks. Therefore, the problem discussed in Example 9 is avoided.

4.3 Other Features

Enumerating Cunits Besides specifying what a cunit does, a developer also needs to tell Perada how to enumerate all cunits so that they cover the entire parameter space for perturbation analysis. Perada then assigns cunits to tasks in epochs. This assignment is important—for instance, in Example 14, if we happen to process less impressive stat records earlier, then pruning will not be effective. By default, Perada evaluates cunits in random order, which works well in many situations because it gives Perada a good sample of perturbations early on. We now discuss how Perada implements this feature efficiently without overburdening the developer.

Recall that cunits are parameterized by cunit parameters. If the number of possible settings is not huge (e.g., with grouping, there will be far fewer cunit parameter settings than perturbations), Perada will take a simple approach. The developer supplies a generator function that enumerates all possible cunit parameter settings for the job. Using this function, Perada produces a list of all settings and permutes the list randomly. It can then assign cunits to tasks and epochs in order from the list; each task gets a sublist.

If the cunit parameter space is large, the approach above becomes inadequate: enumeration and permutation will require lots of memory, and large lists of settings will be sent over the network. Perada offers a more efficient approach that allows parallel enumeration of cunit parameter settings in a random order. To enable this approach, the developer supplies the following functions, for a cunit parameter space with d dimensions: **1)** For each dimension k, $\texttt{size}_k\texttt{()}$ returns the number of possible parameter values for this dimension, and $\texttt{val}_k\texttt{(}i\texttt{)}$ returns the i-th ($0 \leq k < \texttt{size}_k\texttt{()}$) possible value for the dimension. These values can be in arbitrary order. **2)** A Boolean function $\texttt{valid}(v_1, \ldots, v_d)$ tests whether a given combination of parameter values is a valid cunit parameter setting to be considered. A concrete example will be presented later in Example 15. The product of the sets of possible values in individual dimensions defines the *candidate* cunit parameter space $\mathcal{C}^\star$, with size $\prod_k \texttt{size}_k\texttt{()}$. Each element of $\mathcal{C}^\star$ can be labeled a unique integer in $[0, |\mathcal{C}^\star|)$. Using *LCGs* (*linear congruential generators*) with carefully chosen prime moduli and the *leapfrog* method [5], Perada allows all tasks in each epoch to step through a specific random perturbation of all labels (integers in $[0, |\mathcal{C}^\star|)$) in parallel while picking up labels in a round-robin fashion. Thus, all parameter settings are enumerated on the fly (see Appendix A.2 for details).

Instead of relying on Perada's random ordering of cunits and automatic tuning of epochs, developers can also dictate their preferences manually through our API (although we have not found the need to exercise this option in our workloads). We omit the API details because of space constraints.

Declaring Ochoices Oftentimes, developers come up with ideas to optimize their program, e.g., memoization and pruning, but do not know how they will play out for a given workload. Perada allows developers to declare *ochoices*, Boolean flags used to turn on and off specific optimizations in code. When writing the program, developers enclose code blocks implementing a specific optimization with conditional statements that test the corresponding ochoice. The most common uses of ochoices in our programs are for controlling whether to perform memoization using a GlobKV cache, or whether to perform pruning using a SyncSQL cache. We give one example in Example 15.

Perada will explore ochoice settings at run time and automatically pick the setting that maximizes performance; we will discuss that aspect of the Perada optimizer in Section 5.2. Our current implementation of the ochoice API is still rudimentary. We assume that the ochoices are orthogonal, and we can adjust their settings on a per-cunit basis. We have future plans to implement a more sophisticated API that allows specifications of dependency among ochoices and other granularities of adjustment.

4.4 A Complete Example

Example 15 (Congress/vote; implementation in Perada)**.** *We show how to implement the analysis of Example 4 with one Perada job.*

Using grouping, we parameterize each cunit by a pair (d, r) of Democrats and Republicans. The cunit finds all periods (spanning whole months) over which agree% *between d and r is at least τ.*

To enable Perada's default randomized enumeration of cunits (Section 4.3), we simply list all Democrats (Republicans) in $\mathcal{D}$ by id, and define $\texttt{size}_1()$ and $\texttt{val}_1(\cdot)$ ($\texttt{size}_2()$ and $\texttt{val}_2(\cdot)$, resp.) according to this list. We define $\texttt{valid}(d, r)$ to return `true` *if d and r served simultaneously in the same chamber of the Congress.*

A straightforward implementation would retrieve the join (by roll-call-id*) of d and r's votes from $\mathcal{D}$ into memory as a time series, compute* agree% *over all periods, and return those above τ. Suppose the time series has L votes spanning ℓ months. A reasonably efficient implementation can do the in-memory processing in $O(L + \ell^2)$ time (see Appendix A.3 for details). It does not appear that different cunits can benefit from each other's computation.*

A developer with good domain knowledge, however, would note that most legislators vote fairly closely with their own parties, and that the Democratic majority and Republican majority—let us call them $\bar{d}$ and $\bar{r}$, respectively—tend to agree far less than the threshold τ of interest. This insight translates to the following strategy. Before rushing to compare d and r directly, we first check how they vote with their respective party majorities. Intuitively, if both mostly agree with their parties and their parties disagree a lot, the two legislators cannot have high agree% *(see Appendix A.3 for details). At first glance, this strategy makes a cunit do more work: instead of just comparing d vs. r, we would first compare d vs. $\bar{d}$, r vs. $\bar{r}$, and $\bar{d}$ vs. $\bar{r}$, before we have a chance to avoid comparing d vs. r. However, with memoization, this strategy is very appealing. Suppose there are $O(K^2)$ cunits. The result of comparing d vs. $\bar{d}$ (and similarly, r vs. $\bar{r}$) can be cached and reused for all $O(K)$ cunits involving d (and similarly, those involving r); the result of comparing $\bar{d}$ and $\bar{r}$ can be cached and reused for all $O(K^2)$ cunits. Overall, the total number of such comparisons is only $O(K)$, but their results may eliminate many of the $O(K^2)$ direct comparisons.*

To implement this optimization, we add one code block to the cunit before it directly compares d and r. In this code block, we would compare d vs. $\bar{d}$, r vs. $\bar{r}$, and $\bar{d}$ vs. $\bar{r}$—we first look in the GlobKV cache to see if any comparison results are already available; if not, we perform the necessary computation and cache the results. Based on the results, we determine the set $\mathcal{I}$ of candidate periods for which we need to compare d vs. r directly; if $\mathcal{I}$ is empty, the cunit can simply finish.

To allow Perada to turn this optimization on and off automatically, we declare an ochoice. We then enclose the above code block with a conditional statement testing the ochoice. Using another a conditional statement testing the same ochoice, we have the rest of the code in cunit consider either just $\mathcal{I}$ or all periods of interest.

Recall that our goal in this paper is not coming up with the best algorithm for a specific perturbation analysis. Hence, we make no claim of the optimality of our implementation in Example 15. Instead, with this example, we hope to illustrate how Perada can help developers translate their insights quickly into implementation, and relieve them from the burden of tuning the implementation on specific problem instances. For example, the majority-based optimization can easily backfire if τ is low (or if partisan politics end miraculously in the Congress); Perada should be able to detect this situation and turn off the optimization automatically—which brings us to the topic of the next section.

5 Automatic Optimizer

Perada needs to make important decisions on the settings of ochoices and organization of cunits into epochs, yet it has relatively little information about the workload to begin with. Hence, Perada takes the approach of observing, learning, and adapting during execution. This section describes the optimizer in detail.

5.1 Cost Modeling

We focus on three relevant cost components: **1)** Execution time of cunits. **2)** Extra time needed to start a new epoch, which reflects the cost of synchronizing the SyncSQL cache between epochs as well as other overheads. **3)** Effect of output size on future jobs (if any) in the program. We see this effect in CBB/streak (Example 12), where caching affects the number of candidates produced by the first job, which require further processing by the second job. To simplify discussion, assume fixed ochoice settings for now; we will return to the optimization of ochoices in Section 5.2.

Modeling Cunit Execution Time Motivated by the phenomenon of diminishing returns observed in Example 10, we assume that in a serial execution setting, the expected execution time of a cunit in a job is a function of the job *progress*—a number between 0 and 1 representing the fraction of all cunits completed thus far. Intuitively, as more cunits complete and contribute to caches, optimization opportunities such as memoization and pruning increase but eventually plateau. Accordingly, expected execution time should decrease and eventually converge to some minimum. Hence, we choose to fit expected cunit execution time to an offsetted expoential decay function of job progress $t \in [0, 1)$: $C_{\mathfrak{c}}(t) = a + be^{-\lambda t}$. This function form can also handle cases where optimizations are very effective (large λ) or not at all (λ close to 0). In our real workloads, we observe that the actual average cunit execution times follow this trend quite well, as the following examples show. These examples also illustrate an important technicality in how we define, in a parallel setting, the appropriate t for a cunit $\mathfrak{c}$. Intuitively, t should reflect the fraction of cunits whose execution benefits $\mathfrak{c}$—this subtlety is crucial in modeling the non-immediate nature of the SyncSQL caches.

Example 16 (Congress/vote; cunit execution time)**.** *Recall Example 15. We study an execution trace with $n = 16$, $w = 1$, and one job divided into two epochs with $2 : 98$ split of cunits. Only the GlobKV cache is on. We pick one task from each of the two epochs, and show in Figure 4 the execution times vs. t for all cunits in each task (other tasks in the same epoch exhibit a similar pattern). Here, t corresponds to "global" progress. As an example, for a cunit $\mathfrak{c}$ near the end of a task in the first epoch, its t would be close to 2%, even though the task itself only produces $1/n$ of this progress. The justification is that $\mathfrak{c}$ benefits from almost all cunits in the epoch (including those from concurrent tasks) through the GlobKV cache, where information exchange is immediate.*

Note there is significant variance in the execution times of individual cunits. Slowest are those cunits that perform comparisons with the party majorities for the first time; fastest are those using previously cached comparison results to eliminate most periods from consideration. Over time, density shifts to lower execution times. Therefore, the rolling average of execution times decreases. Overall, we see that the fitted $C_{\mathfrak{c}}(t)$ captures this trend well.

Example 17 (MLB/dom; cunit execution time)**.** *Recall Example 14. We study an execution trace with $n = 4$, $w = 1$, and one job divided into two epochs with $2 : 98$ split of cunits. Only the SyncSQL cache is on this time. In the same fashion as Example 16, for one task from each of the two epochs, Figure 4 shows the execution times vs. t for all cunits in the task.*

However, t is determined differently here. For example, consider again a cunit $\mathfrak{c}$ at the end of a task in the first epoch. Its t would be only be about 0.5%, corresponding to those cunits that executed

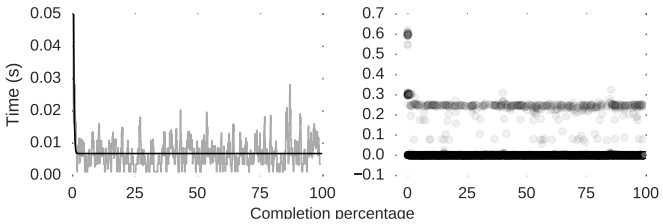

Figure 3: Cunit execution times over the course of running Congress/vote with $\tau = 45\%$. Here, $n = 16$ and $w = 1$, there are two epochs, and the GlobKV cache is on. See Section 6 for details on the experimental setup.

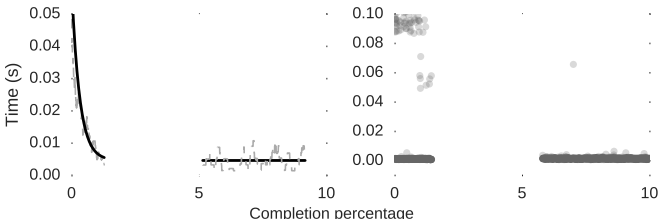

Figure 4: Cunit execution times over the course of running MLB/dom with $\{$*hits-allowed*, *strikeouts*, *earned-run-avg*$\}$ and $\kappa = 5000$. Here, $n = 4$ and $w = 1$, there are two epochs, and the SyncSQL cache is on. See Section 6 for details on the experimental setup.

before $\mathfrak{c}$ in the same task. *The reason is that the local SyncSQL database does not yet see updates from other concurrent tasks. As soon as the epoch completes, however,* t *leaps forward, because synchronization makes information from all tasks visible. Thus, Figure 4 shows a gap in* t *between data from the two epochs.*

Despite this difference, we again see that $C_{\mathfrak{c}}(t)$ *fits the trend in cunit execution times quite well.*

Each Perada task monitors the execution times of its cunits and maintains a small summary sufficient for fitting $C_{\mathfrak{c}}(t)$. Perada takes care in aligning each cunit with the appropriate t as illustrated in the examples above (using function $T(\cdot)$ further discussed below). At the end of an epoch (if it is not the last in a job), the Perada master collects these summaries and fits (or refits) $C_{\mathfrak{c}}(t)$. In the case of on-demand cunit enumeration (Section 4.3), Perada also observes the selectivity of `valid(·)`, and is able to produce an estimate for N, the total number of cunits in a job, by the end of its first epoch.

Using $C_{\mathfrak{c}}(\cdot)$, it is straightforward to derive an estimate for the execution time of an epoch $\mathfrak{e}$ as $C_{\mathfrak{e}}(s, \delta)$, where s is the progress before $\mathfrak{e}$ starts, and δ is the amount of progress to be made by $\mathfrak{e}$. Since the epoch has nw tasks in w waves, each task is responsible for $\frac{\delta N}{nw}$ cunits. Perada estimates the execution time of a task in the j-th wave as $C_{\mathfrak{t}}(s, \delta, j) = \sum_{i=1}^{\frac{\delta N}{nw}} C_{\mathfrak{c}}(s + T(\delta, i, j))$, where $T(\delta, i, j)$ helps calculate the appropriate t for the i-th cunit in the task: if the cunits use the SyncSQL cache, this calculation follows the reasoning in Example 17; otherwise, Example 16 is followed. Given the exponential form of $C_{\mathfrak{c}}(\cdot)$, it is straightforward to simplify the summation and derive a closed-form formula for $C_{\mathfrak{t}}(s, \delta, j)$. We omit the details. Then, the estimated execution time of the epoch is simply $C_{\mathfrak{e}}(s, \delta) = \sum_{j=1}^{w} C_{\mathfrak{t}}(s, \delta, j)$.

Note that Perada's execution time modeling relies on only high-level execution statistics; it does not require detailed knowledge about the program—developers can write in a general-purpose programming language, and need not tell Perada what constitutes cache hits or successful pruning. So far, we have found our simple approach to be effective—we attribute its success to our randomized processing order of cunits, which fits the assumption of diminishing returns nicely, and to our limited use of $C_{\mathfrak{c}}(\cdot)$ (only in summation), which reduces variances in estimates.

Modeling New Epoch Overhead Following an epoch $\mathfrak{e}$ that progresses the job from s to $s + \delta$, Perada assumes that the extra time $C_{\dagger}(s, \delta)$ involved in having an new epoch after $\mathfrak{e}$ is some fixed overhead plus a synchronization component proportional to the total amount of "effort" spent in $\mathfrak{e}$. (Note that we charge all cost of synchronization—some of which is carried out at the end of $\mathfrak{e}$—to $C_{\dagger}(s, \delta)$ instead of $C_{\mathfrak{e}}(s, \delta)$.) Intuitively, more effort implies a higher chance of producing new information worthwhile caching, which translates to a higher synchronization cost. As the caches become more effective, less effort is required in evaluating cunits, and less is synchronized. Perada defines the effort of $\mathfrak{e}$ as the total execution time over all tasks, so $C_{\dagger}(s, \delta)$ has the form $a + bnC_{\mathfrak{e}}(s, \delta)$, where a and b need to be learned.

When the first epoch in a job begins, Perada observes the start-up time, which serves as an estimate for a. Perada also estimates a per-byte broadcast cost, by noting the amount of data broadcasted (including $\mathcal{D}$ but no cached data at this point) and time taken. When the first epoch ends, Perada knows 1) the actual effort of the epoch, 2) the elapsed time attributed to computing deltas and applying them, and 3) the amount of cached data to be broadcasted, which provides a projection for the additional time needed to broadcast cached data. Taken together, these quantities give another data point to fit $C_{\dagger}(s, \delta)$ with. Perada revises $C_{\dagger}(s, \delta)$ at the end of every subsequent epoch (except the last in the job).

Modeling Output Impact on Subsequent Jobs As noted either, in a program with multiple jobs, optimization decisions made for one job could affect its output size, which in turn affects the cost of subsequent jobs (Example 12). To account for this effect, Perada charges a cost to each unit of output produced, estimated as γa, where a is the average cunit execution time for the current job (estimated from $C_{\mathfrak{c}}(\cdot)$ as discussed earlier), and γ is a discount factor. We use $\gamma = 1/5$, reflecting our assumption that subsequent jobs usually require less processing. Perada further assumes that the output size of each epoch is proportional to its effort (as defined for $C_{\dagger}(\cdot)$); the coefficient b is estimated by monitoring the amount of effort and the actual output size so far in the job. In sum, we model the effect of output from an epoch that progresses from s to $s + \delta$ as $C_{\mathfrak{o}}(s, \delta) = \gamma abnC_{\mathfrak{e}}(s, \delta)$.

5.2 Ochoice Probing

For simplicity, Section 5.1 has assumed fixed ochoice settings. In reality, Perada does not know the appropriate settings in advance, so it explores them at run time to pick the optimal setting. Currently, Perada employs a simple strategy that works fine as long as the number of ochoices per job is low. Suppose there are h available ochoices. Formally, an *ochoice setting* is a bit string of length h; there are 2^h possible settings. Recall from Section 4.3 that each cunit can use its own ochoice setting. In the first epoch of a job, Perada focuses on exploring: each task executes an equal fraction of its cunits using each of the possible 2^h settings. During each epoch, Perada tracks the cunit execution times by their ochoice settings, and at the of the epoch, it fits a $C_{\mathfrak{c}}(\cdot)$ for each setting. Suppose the setting that gives the fastest execution times is $\mathbf{b}^{\star}$. In the next epoch, each task would execute the majority (we currently use 75%) of the cunits using $\mathbf{b}^{\star}$, and divide the remaining cunits equally among the other $2^h - 1$ settings. This strategy allows Perada to continue monitoring all possibilities and detect when a different setting may become optimal later. Of course, if the next epoch happens to the last in the job, Perada will instead go "all-in" with $\mathbf{b}^{\star}$.

A few more intricacies need to be handled when we allow a task to run its cunits with a mix of ochoice settings. Let f denote the *configuration* for this mix, where $f(\mathbf{b})$ is the fraction of cunits executing with ochoice setting $\mathbf{b}$ in the current epoch. We need to modify the function $T(\cdot)$ described in Section 5.1 to account for f—subtleties arise because cunits using settings that turn off caching using GlobKV should effectively decrease $T(\cdot)$ for cunits that use this type of cached data. The summation in the definition of $C_{\mathfrak{t}}(\cdot)$ (Section 5.1) also needs to be modified to consider configuration f and use the individually fitted $C_{\mathfrak{e}}(\cdot)$ for each setting with the modified $T(\cdot)$. We omit the details.

5.3 Epoch Organization

Finally, we turn to the discussion of how to organize cunits in a job into epochs to provide a good trade-off between cost and benefit of synchronization. Perada always uses a small epoch to start a job—by default we set it to stop at 2% progress. This brief exploration period helps Perada gain initial knowledge of various cost functions. At the end of an epoch finishing at progress point s, if $s < 1$ (i.e., the job still has work remaining), Perada decides what to do next as follows.

1. Using $C_{\mathfrak{e}}(\cdot)$ fitted for each ochoice setting, Perada finds the ochoice setting $\mathbf{b}^\star$ that minimizes the time $C_{\mathfrak{e}}(s, 1-s)$ if we execute all remaining work in one epoch where all cunits use the given ochoice setting. (Note that this objective also minimizes output by our assumption in Section 5.1.)
2. Perada considers the following alternatives:
   (a) Execute all remaining work in one epoch where all cunits use ochoice setting $\mathbf{b}^\star$. The total cost is $C_{\mathfrak{e}}(s, 1-s) + C_{\mathbf{o}}(s, 1-s)$. (We do not include the cost of starting this epoch because all alternatives need to start at least one epoch and pay the same cost.)
   (b) Execute all remaining work in two epochs $\mathfrak{e}_1$ and $\mathfrak{e}_2$, first from s to $s+\delta$ and then to 1. As described in Section 5.2, we execute $\mathfrak{e}_1$ with configuration f_1, where 75% of the cunits use $\mathbf{b}^\star$ and the remaining are divided evenly among other ochoice settings; we execute $\mathfrak{e}_2$ with configuration f_2, where all cunits use $\mathbf{b}^\star$. The total cost is the sum of $C_{\mathfrak{e}}(s, \delta) + C_{\mathfrak{e}}(s+\delta, 1-s-\delta)$ (executing $\mathfrak{e}_1$ and $\mathfrak{e}_2$), $C_{\dagger}(s, \delta)$ (starting $\mathfrak{e}_2$), and $C_{\mathbf{o}}(s, \delta) + C_{\mathbf{o}}(s+\delta, 1-s-\delta)$ (effect of output from $\mathfrak{e}_1$ and $\mathfrak{e}_2$).

   Perada picks the alternative with the lowest total cost, which involves searching through possible δ for the 2-epoch alternative.

We note that this optimization procedure cuts several corners. First, Perada optimizes the ochoice setting first (by assuming a single epoch for the remaining work); considering ochoice setting and epoch organization jointly would yield a bigger search space. Second, in Step 2 above, Perada does not enumerate alternatives using more than two epochs for the remaining work; considering them would significantly expand the search space.

Perada chose to cut these corners for several reasons. Most importantly, we want to keep the optimization time low. Also, it may be unwise to overdo advance planning as cost models could continue evolving during execution. Finally, in reality, Perada invokes the above procedure at the end of every epoch, so there is opportunity for further reorganization. Even if we decide to go with two epochs $\mathfrak{e}_1$ and $\mathfrak{e}_2$ for now, as we come to the end of $\mathfrak{e}_1$, we may decide to divide $\mathfrak{e}_2$ further; we will see some real examples with more than two epochs in Section 6.

6 Experiments

Hardware and Software Setup The experiments were performed on Google Compute Engine using varying number of machines with instance type `n1-standard-4` (4 virtual CPUs and 15GB memory per node). Thus, in this section, the number n of slots available for parallel execution is 4 times the number of machines. All machines run Ubuntu 14.04, Spark 1.5, Python 2.7, and Pandas .16. The local databases use SQLite 3.7.13, while the GlobKV cache uses Redis 3.0.5. Perada itself is implemented in Python.

Workloads We experimented with examples used throughout this paper, which represent real tasks in sports and politics journalism:

- MLB/dom/bat: *Example 5 with batting-related stats* $\mathcal{M} = \{\text{hits}, \text{home-runs}, \text{stolen-bases}\}$ *and maximum* domcount $\kappa = 50$. *The* player *parameter is restricted to batters only. The database includes all yearly statistics from* seanlahman.com.
- MLB/dom/pitch: *Example 5 with pitching-related stats* $\mathcal{M} = \{\text{hits-allowed}, \text{strikeouts}, \text{earned-run-avg}\}$ *and maximum* domcount $\kappa = 50$. *The* player *parameter is restricted to pitchers only. The database is the same as in MLB/dom/bat.*
- Congress/vote: *Example 4 with minimum* agree% $\tau = 45$. *The database includes all votes and legislators in the U.S. Congress during 2002–2014, from* govtrack.us.
- CBB/streak*: Example 3 with maximum* domcount $\kappa = 25$. *The database includes all historical per-game statistics of Duke Men's basketball team.*

The following table list some relevant descriptive statistics about these workloads. Here, $|\mathcal{D}|$ denotes the number of rows in the input database, $|P|$ denotes the number of possible parameter settings, and "# cunits" is the number of cunits in the first job (recall that CBB/streak uses two Perada jobs; others use one).

Workload	$\lvert\mathcal{D}\rvert$	$\lvert P\rvert$	# cunits
MLB/dom/bat	656,214	598,722	598,722
MLB/dom/pitch	656,214	277,716	277,716
Congress/vote	8.8×10^6	6.0×10^6	318,833
CBB/streak	23,227	2.6×10^6	4,857

Note that MLB/dom/bat has a larger $|P|$ than MLB/dom/pitch because there are more batters than pitchers. For Congress/vote and CBB/streak, because they use grouping, the number of cunits is significantly lower than that of parameter settings.

6.1 Overall Comparison

For each workload, we run its Perada-based implementation with varying degree of parallelism ($n = 8, 16, 32$) and compare its performance against three alternative implementations on top of a database system. The first, *SQL/I(ndependent)*, considers each perturbation independently using a SQL query. The second, *SQL/C(ombined)*, divides the parameter space among the n slots, and considers all perturbations assigned to a slot using a single SQL query (with heavy use of `WITH` clauses)—the goal here is to let the database system optimize all perturbations as a whole. Lastly, *SQL+M(QO)* is a hand-coded implementation approximating what a database system armed with state-of-the-art multiple-query processing (MQO) techniques (further discussed in Section 7) might do in the best case.

We implemented SQL/I, SQL/C, and SQL+M on PostgreSQL 9.4, because its optimizer is more capable than SQLite. We spent reasonable effort tuning these implementations. We replicated the database and created all applicable indexes. For MLB/dom and Congress/vote, we wrote queries in SQL/I and SQL/C to also apply post-processing (SQL/C required considerable tweaking because the PostgreSQL optimizer is less capable of pushing selections down into temporary tables defined using `WITH` [2]). For CB-

	$n = 8$				$n = 16$				$n = 32$			
Workload	**Perada**	**SQL/I**	**SQL/C**	**SQL+M**	**Perada**	**SQL/I**	**SQL/C**	**SQL+M**	**Perada**	**SQL/I**	**SQL/C**	**SQL+M**
MLB/dom/bat	257.5	> 8hr	> 8hr	4019.4	209.1	16405.2	15439.7	1989.8	157.9	8202.4	7668.1	1058.4
MLB/dom/pitch	153.1	25428.9	6524.2	3704.3	122.5	12706.2	2676.0	1937.1	78.0	6367.5	1269.0	961.8
Congress/vote	1103.5	> 8hr	> 8hr	4270.1	599.3	22312.1	> 8hr	2587.2	333.8	11101.4	> 8hr	1293.2
CBB/streak	82.2	145.0	622.8	199.8	71.6	78.5	311.4	85.4	58.9	36.2	155.7	56.6

Table 1: Execution times of Perada vs. other alternatives (in seconds unless otherwise noted) across workloads, with varying degrees of parallelism.

B/streak, both SQL/I and SQL/C require two steps: the first step computes all candidate streaks in parallel, and after we collect the set of all candidates, the second step computes *domcount* for candidates in this set in parallel.

To implement SQL+M, we identified techniques from the MQO literature most suitable for each workload and implemented them by hand, since most of them are not supported by existing database systems. For MLB/dom and CBB/streak (step 2), we ensured sharing of common subexpressions to achieve a similar effect as memoization; however, we note that existing MQO techniques are unlikely to find this optimization automatically (see Appendix A.4 for a detailed explanation). For Congress/vote and CBB/streak (step 1), we adapted techniques for sharing sliding-window aggregates from [3] (again, see Appendix A.4 for details); we had to implement this part of the processing in Python. Overall, we believe SQL+M approximates what existing MQO techniques could achieve in the best case.

We present results comparing Perada with these alternatives in Table 1. In all cases (except one to be discussed further below), Perada is the winner, and most of the time by a huge margin—often more than an order of magnitude. Among the rest, SQL+M tends to be the fastest and SQL/I the slowest, but there are other considerations. First, as discussed above, we gave SQL+M more advantage than others, and it is not viable in practice currently. SQL+M and SQL/C were more difficult to develop than SQL/I, and SQL/C has some performance quirks—in some cases, it materialized massive temporary tables dozens of GB in size. Finally, while SQL+M and SQL/C may take less time to produce all answers, SQL/I can produce some answers earlier (Perada also produces individual answers as early as possible, besides producing all answers the fastest).

Even though SQL+M was able to benefit from parallelization, grouping, and memoization (thanks to manual tweaks and implementation of specialized algorithms), it performs much worse than Perada. We attribute Perada's advantage to its ability to prune based on the post-processing query (Examples 8, 9, and 15). Lacking knowledge of the post-processing query, MQO techniques cannot exploit pruning by themselves. Grouping all perturbation and post-processing queries into a single query does not help either, as database systems currently lack support for pruning optimization.

In Table 1, there is one case—CBB/streak when $n = 32$—where the simple approach of SQL/I overtakes Perada, and SQL+M also is slightly better. Here, the database and the amount of work per execution slot are so small that the various overheads of Perada start to become significant. Nonetheless, the gap is small, in contrast to consistently large gains that Perada achieves for bigger workloads.

Finally, based on our experience implementing various alternatives, we observe that while SQL is declarative, it can be awkward to use when coding complex perturbation analysis, and worse, when using it to implement (or "coaxing" it to enable) specific optimizations. Overall, we found Perada's Python/database combination easier to develop with than SQL/C and SQL+M.

6.2 Benefit and Choice of Caching

We now study the benefit of caching and how effective Perada is at enabling appropriate caching-based optimizations (through its automatic setting of ochoices). Here, we compare Perada with three variants where we turn off the optimizer and hard-code all ochoice settings: "SyncSQL only" and "GlobKV only" always enable optimizations using the respective cache while disabling those using the other cache; "no caching" disables all caching-based optimizations. We make these variants use the same epoch sizes as Perada. Table 2 summarizes the results for $n = 16$.

Across workloads, we see that turning on caching helps (except that our Congress/vote implementation does not use the SyncSQL cache). A more interesting observation is that the benefit of each cache type depends on the specific workload. For example, consider MLB/dom/bat and MLB/dom/pitch, which apply the same analysis (and code) to the same dataset, and differ only in what stats they compare. The GlobKV cache helps MLB/dom/bat far more than it does MLB/dom/pitch ($10\times$ vs. $4\times$ speedup compared with no caching). The reason is that all stats in MLB/dom/bat are integer-valued, so the possible number of stat value combination for this case is far less than that for MLB/dom/pitch, where the floating-point *earned-run-avg* has far more possible values. For both workloads, the SyncSQL cache is more effective than the GlobKV cache, though the opposite is true for CBB/streak. Regardless of who the winner is, Perada's performance is close to or even better than the winner—which is good considering that Perada does not know who the winner is in advance.

Table 3 further shows, for the workloads in Table 2, which caching strategy Perada chooses eventually (in the last epoch of each job; recall that CBB/streak has two). Here, we see how Perada is able to achieve its performance in Table 2. As an example, for CBB/streak, Perada learns that in its second job using SyncSQL and GlobKV together is better than using either alone; this observation allows Perada to beat both "SyncSQL only" and "GlobKV only" in Table 2. Turning on both caches is not always optimal, however. As another example, for MLB/dom/pitch, Perada recognizes that SyncSQL's benefit overshadows GlobKV's, and that the marginal gain of adding GlobKV is not worth its overhead (because of the large stat value space). Therefore, Perada decides to go with SyncSQL alone; the extra time over "SyncSQL only" in Table 2 is the price Perada paid for acquiring this knowledge (in the first epoch).

Our next experiment delves into CBB/streak, a two-job workload, to study how job output size affects the rest of execution, and how caching influences this effect. Figure 5 plots two group of bars. For the first group, we let Perada optimize the first job; for the second, we manually turn off all caching in the first job. The first bar in each group shows the size of the first job's output, which serves as the input to the second job. We see that Perada's caching significantly reduces the output size, because SyncSQL is very effective in pruning non-promising streaks from further consideration. Then, in the second job, we again try two strategies: one where we manually turn off caching, and one where Perada decides. The resulting end-to-end execution times for these two strategies are shown as the second and third bars in each group. From Figure 5, we see that a smaller intermediate output translates to proportionally faster total execution time, even if the second job makes a poor (no) caching decision. A bigger immediate output would have had a disastrous impact, but in this case it is significantly mitigated by Perada's effective caching in the second job. Nonetheless, we achieve the best

Workload	Perada	SyncSQL only	GlobKV only	No caching
MLB/dom/bat	209.1	170.3	1081.3	11945.6
MLB/dom/pitch	122.5	116.1	2080.4	8028.7
Congress/vote	599.3	3043.8	581.3	3043.8
CBB/streak	71.6	130.1	80.1	2634.0

Table 2: Execution times of Perada vs. variants with explicitly set ochoices for caching-based optimizations. Here $n = 16$.

Workload	Strategy
MLB/dom/bat	SyncSQL + GlobKV
MLB/dom/pitch	SyncSQL
Congress/vote	GlobKV
CBB/streak job 1	SyncSQL
CBB/streak job 2	SyncSQL + GlobKV

Table 3: Perada's ochoice setting in the last epoch of each job for workloads in Table 2.

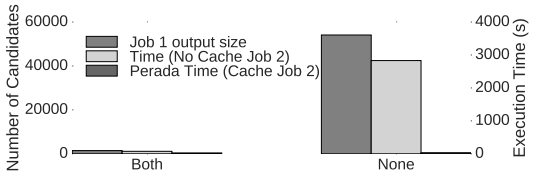

Figure 5: Comparison of intermediate output size and total execution time for CBB/streak across various options as explained in Section 6.2.

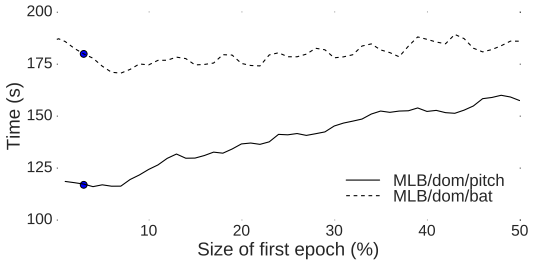

Figure 6: Execution times of MLB/dom/bat and MLB/dom/pitch vs. initial epoch size, with the number of epochs is fixed at 2. Perada's default 2% setting is marked.

performance overall only with intelligent caching decisions in both jobs, in a way cognizant of the effect of intermediate output sizes.

6.3 Effect and Choice of Epochs

We now examine study how epochs affect performance and how effective Perada is in choosing epoch sizes. We start with an experiment testing how the size of the initial epoch affects the overall performance (recall that Perada defaults it to 2%). This experiment also illustrates the trade-off involved in the SyncSQL cache synchronization. In Figure 6, we show the total execution times of MLB/dom/bat and MLB/dom/pitch as functions of the initial epoch size, while forcing Perada to always use two epochs. Both functions follow a similar trend—they first decrease, and then increase—which we also observe in other workloads that use SyncSQL. Intuitively, if the initial epoch is too small, the resulting SyncSQL cache after synchronization will not be effective enough for the second epoch. On the other hand, if the initial epoch is too big, then the first epoch would be suboptimal. The initial epoch sizes that yield fast execution times tend to be small, which justifies our default setting of 2%.

Next, let us consider the effect of the number of epochs. For this experiment, we use MLB/dom/bat. After the the initial epoch, Perada decides to go with one epoch for all remaining work; however, we try alternatives with different number of equal-sized epochs. Figure 7 shows the cumulative execution times of these alternatives: each point on a plot shows the time elapsed since the beginning of the job when each epoch ends. For comparison, we also show Perada with a single epoch with the optimal ochoice setting. In this case, single epoch performs better than Perada's two epochs, but only slightly; furthermore, Perada does not know the optimal ochoice setting in advance, so this small overhead is the price to pay for acquiring this knowledge. As for alternatives with more epochs, they underperform Perada by a big margin, as they suffer from the overhead of synchronization and starting new epochs.

The above experiment does not imply that one or two epochs always work best. Let us now consider a third experiment. In Section 4.3, we described how developers can override Perada's cunit enumeration method. Figure 8 shows a scenario where MLB/dom/pitch with $\kappa = 5000$ receives a parameter allocation that is not randomized. Because of lack of randomization, some tasks have underperforming caches in the initial epoch, yielding a poor pruning rate and a slowly improving model in the optimizer. Here, we see that the single-epoch alternative performs drastically slower than the other two due to ineffective local SyncSQL databases, whereas the three-epoch alternative outperforms the two-epoch one (albeit slightly). In this case, the ability of Perada to use multiple epochs acts as a "guard" against such situations, by providing a larger sample of executions to its cache and optimizer before executing the final epoch.

6.4 Scaling with Workload Size

Our next experiment tests the ability of Perada to scale with the size of its workload. As we vary n, the degree of parallelism, we also adjust the total amount of work accordingly such that $1/n$ of this amount remains roughly constant. As the parameter spaces of the workloads in our early experiments are already quite comprehensive, we start with them for our biggest n, and downsample them to create smaller workloads (instead of inventing unrealistic parameter settings to grow them further). Figure 9 shows the execution times for varying n and workload size. We see that Perada scales well across workloads. As expected, for most of workloads, the execution times follow an increasing trend, as a bigger cluster leads to higher overhead (such as synchronization). The good news is that this growth is generally small; the only exception, CBB/streak, is a very light workload even at its full size. Interestingly, one workload, MLB/dom/pitch, sees a decreasing trend. We theorize that with a larger n, more entries become available in the SyncSQL cache after the first epoch, making the cache more effective. MLB/dom/bat does not exhibit this behavior because its highly effective GlobKV cache dilutes this benefit.

7 Related Work

The problem of efficiently evaluating a large number of "similar" or "related" queries has been considered in many different contexts, such as multiple-query optimization (MQO) [18], scalable continuous query processing [4], and keyword search in relational databases [23] (the references here are intended as starting points of literature search and are by no means complete). Various techniques have been developed to share common subexpressions and group-process filters and aggregates. Some of the high-level ideas in Perada can be seen in these contexts: e.g., grouping and memoization in multiple-query optimization and scalable continuous query processing, and pruning in keyword search. However, perturbation analysis workloads are different in many ways: our queries are identical except for their parameter settings; our query template can be far more complex and the number of instantiations is huge; finally, we also have a potentially complex post-processing step over all query results. These differences lead us to rather different foci and techniques. For example, compared with MQO,

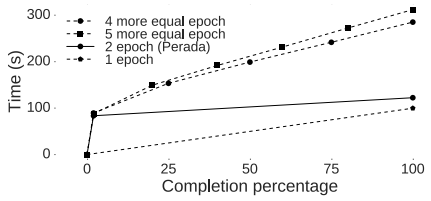

Figure 7: Cumulative execution times (by epoch) of MLB/dom/bat, with varying number of epochs.

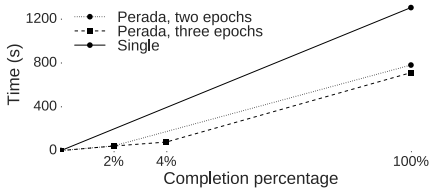

Figure 8: Cumulative execution times (by epoch) of MLB/dom/pitch with $\kappa = 5000$. Note that the horizontal axis has a logarithmic scale.

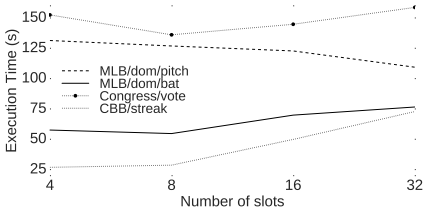

Figure 9: Execution times of Perada for different workloads, as their sizes increase linearly with the degree of parallelism.

Perada's pruning based on the post-processing query is a unique and powerful optimization that underpins the significant advantage over SQL+M illustrated in Section 6.1. While we believe Perada's current suite of techniques captures the most important optimizations for perturbation analysis, there is certainly room to continue improving our toolbox with techniques from other contexts.

Efficient algorithms have been developed for many problems that can be cast as perturbation analysis. A (non-comprehensive) list of examples include *iceberg queries* [9], *one-of-the-few objects* [21], *prominent streaks* [24], as well as fact-checking *window aggregate comparison* and *time-series similarity* claims [22]. However, they focus on very specific query templates. As discussed in Section 1, Perada aims instead at enabling general, ad hoc perturbation analysis for common developers, who may not have the expertise or time to develop specialized algorithms.

There have been a few notable efforts at solutions for more general query templates. *Query flocks* [20] generalize associate-rule mining to a broader class of parameterized conjunctive queries with monotone filters. *Searchlight* [12] supports searching for data "regions" (multi-dimensional subarrays) with desired properties, which can be seen as parameterized multi-dimensional selection followed by aggregation. Both rely heavily on pruning, and Searchlight also exploits parallelism. Perada is more general, and a developer can implement similar pruning ideas in Perada; the price for this generality is the extra development effort and potentially some missed opportunities compared with these more specialized solutions.

There is an interesting connection between query perturbation analysis and work on *explanations* in databases (surveyed in [14]). Problems such as searching for the best predicate *interventions* [17] or *refined queries* [19] to explain unexpected results can be seen as searching a space of possible query perturbations for those returning expected results. Another interesting but less obvious connection is with probabilistic databases [8]. Given a query template, it may be possible to regard the parameter space P as a distribution for an uncertain "parameter setting tuple," such that its "join" with the query template would yield the the distribution of perturbations. Probabilistic query evaluation techniques, such as those of *MCDB* [11], may be applicable to some types of post-processing queries in perturbation analysis, especially when approximation is allowed. We plan to investigate these connections as future work.

8 Conclusion and Future Work

In this paper, we have introduced a novel system called Perada for perturbation analysis of database queries. Perada provides a parallel execution framework tailored towards processing a large number of perturbations. Perada's flexible API and dual-cache support make it easy to implement general, ad hoc perturbation analysis with a variety of optimizations such as grouping, memoization, and pruning, and hide the complexity of concurrency and failures. Perada's automatic optimizer observes, learns, and adapts during execution, eliminating the need for manual performance tuning. Experiments with real computational journalism workloads demonstrate Perada's advantage over alternatives solutions.

While this paper focuses on computational journalism applications, Perada and its techniques apply to many other domains where query perturbation analysis is useful. Perada represents our first step towards supporting perturbation analysis of database queries. Although Perada has greatly simplified implementation and tuning, it is still not fully declarative system. To further improve usability, we are actively working on automatic derivation of memoization and pruning opportunities from declarative specifications. Other vectors of future research include more dynamic, finer-grained optimization and adaptation, as well as a number of interesting connections to other database research problems as discussed in Section 7. In conclusion, we believe that perturbation analysis is a practically important and technically interesting modern database feature that deserves more research.

References

[1] SQLite. URL `http://sqlite.org/`.

[2] PostgreSQL 9.4.6 documentation. 2016. URL `http://www.postgresql.org/docs/9.4/static/queries-with.html`.

[3] Arvind Arasu and Jennifer Widom. Resource sharing in continuous sliding-window aggregates. In *Proceedings of the 2004 International Conference on Very Large Data Bases*, pages 336–347, Toronto, Canada, August 2004.

[4] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 379–390, Dallas, Texas, USA, May 2000.

[5] Paul D. Coddington. Random number generators for parallel computers. Technical Report 13, Northeast Parallel Architecture Center, 1997.

[6] Sarah Cohen, James T. Hamilton, and Fred Turner. Computational journalism. *Communications of the ACM*, 54(10): 66–71, 2011.

[7] Sarah Cohen, Chengkai Li, Jun Yang, and Cong Yu. Computational journalism: A call to arms to database researchers. In *Proceedings of the 2011 Conference on Innovative Data Systems Research*, Asilomar, California, USA, January 2011.

[8] Nilesh N. Dalvi, Christopher Ré, and Dan Suciu. Probabilistic databases: Diamonds in the dirt. *Communications of the ACM*, 52(7):86–94, 2009.

[9] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D. Ullman. Computing iceberg queries efficiently. In *Proceedings of the 1998 International Conference on Very Large Data Bases*, New York City, New York, USA, August 299–310.

[10] The Apache Software Foundation. Apache Spark: Lightning-fast cluster computing. URL `http://spark.apache.org/`.

[11] Ravi Jampani, Fei Xu, Mingxi Wu, Luis Leopoldo Perez,

Chris Jermaine, and Peter J. Haas. The Monte Carlo database system: Stochastic analysis close to the data. *ACM Transactions on Database Systems*, 36(3):18, 2011.

[12] Alexander Kalinin, Ugur Çetintemel, and Stanley B. Zdonik. Searchlight: Enabling integrated search and exploration over large multidimensional data. *Proceedings of the VLDB Endowment*, 8(10):1094–1105, 2015.

[13] Redis Labs. Redis. URL `http://redis.io/`.

[14] Alexandra Meliou, Sudeepa Roy, and Dan Suciu. Causality and explanations in databases. *Proceedings of the VLDB Endowment*, 7(13):1715–1716, 2014.

[15] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. Progressive skyline computation in database systems. *ACM Transactions on Database Systems*, 30(1):41–82, 2005.

[16] A. Remington. Social science done on deadline: Research chat with ASU's Steve Doig on data journalism, 2014. URL `http://goo.gl/a55LDT`.

[17] Sudeepa Roy and Dan Suciu. A formal approach to finding explanations for database queries. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 1579–1590, Snowbird, Utah, USA, June 2014.

[18] Timos K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.

[19] Quoc Trung Tran and Chee-Yong Chan. How to ConQueR why-not questions. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 15–26, Indianapolis, Indiana, USA, June 2010.

[20] Shalom Tsur, Jeffrey D. Ullman, Serge Abiteboul, Chris Clifton, Rajeev Motwani, Svetlozar Nestorov, and Arnon Rosenthal. Query flocks: A generalization of association-rule mining. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 1–12, Seattle, Washington, USA, May 1998.

[21] You Wu, Pankaj K. Agarwal, Chengkai Li, Jun Yang, and Cong Yu. On "one of the few" objects. In *Proceedings of the 2012 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1487–1495, Beijing, China, August 2012.

[22] You Wu, Pankaj K. Agarwal, Chengkai Li, Jun Yang, and Cong Yu. Toward computational fact-checking. *Proceedings of the VLDB Endowment*, 7(7):589–600, 2014.

[23] Jeffrey Xu Yu, Lu Qin, and Lijun Chang. Keyword search in relational databases: A survey. *IEEE Data Engineering Bulletin*, 33(1):67–78, 2010.

[24] Gensheng Zhang, Xiao Jiang, Ping Luo, Min Wang, and Chengkai Li. Discovering general prominent streaks in sequence data. *ACM Transactions on Knowledge Discovery from Data*, 8(2), June 2014.

APPENDIX

A Additional Details

A.1 Incremental Computation for Example 8

Given (x', y'), let $(x^\star, y^\star, c^\star)$ be the entry with the maximum $c^\star$ among those remembered where $(x^\star, y^\star)$ dominates (x', y'). Then, to compute *domcount* for (x', y') on $\mathcal{D}$, there is no need to consider those dominating both $(x^\star, y^\star)$ and (x', y'), because we already know there are $c^\star$ of them. To obtaining the remaining count, we need to issue two rectangular counting queries to cover the "L"-shaped space dominating (x', y') but not $(x^\star, y^\star)$.

A.2 On-the-Fly Enumeration of Cunits

Given a label, a task converts it to the corresponding candidate cunit parameter setting by first computing the index of its value along each dimension k, and then calling $\texttt{val}_k(\cdot)$ to recover the value. If the candidate setting is not valid according to `valid(·)`, the task simply skips to the next label designated for it. Thus, all parameter settings are enumerated on the fly. This method works well as long as $\texttt{valid}(\cdot)$ is not overly selective (i.e., a reasonable fraction of the candidates are valid). While the LCG/leapfrog combination has well-documented issues for more demanding applications [5], it works very well in our case, providing acceptable randomness while being extremely efficient to operate distributedly.

A.3 Details for Example 15

On the $O(L + \ell^2)$ algorithm. In more detail, suppose the time series span the (year-)months $m_1, \ldots, m_\ell$. Let $\text{agree}(d, r, \imath, \jmath)$ denote the number of times that d and r agree during $[m_\imath, m_\jmath]$, and let $\text{total}(d, r, \imath, \jmath)$ denote the number of times that d and r both voted during $[m_\imath, m_\jmath]$. We can make one pass over the time series to compute, for each $\jmath \in [1, \ell]$, quantities $\text{agree}(d, r, 1, \jmath)$ and $\text{total}(d, r, 1, \jmath)$ in time $O(L)$. Then, for each period $[m_\imath, m_\jmath]$ (there are $O(\ell^2)$ of them), we can compute *agree%* as $\frac{\text{agree}(d,r,1,\jmath)-\text{agree}(d,r,1,\imath-1)}{\text{total}(d,r,1,\jmath)-\text{total}(d,r,1,\imath-1)}$.

On pruning using comparisons with party majority. More precisely, for any period $[m_\imath, m_\jmath]$, we have the following (with $\imath, \jmath$ omitted from the arguments to agree() and total() for brevity): $\text{agree}(d, r) \le \min\{\text{agree}(d, \bar{d}), \text{agree}(r, \bar{r}), \text{agree}(\bar{d}, \bar{r})\} + (\text{total}(d, \bar{d}) - \text{agree}(d, \bar{d})) + (\text{total}(r, \bar{r}) - \text{agree}(r, \bar{r}))$; and $\text{total}(d, r) \ge \text{total}(d, \bar{d}) + \text{total}(r, \bar{r}) - \text{total}(\bar{d}, \bar{r})$. From the upper bound on $\text{agree}(d, r)$ and the lower bound on $\text{total}(d, r)$, we can then derive an upper bound on $\textit{agree\%} = \text{agree}(d, r)/\text{total}(d, r)$.

A.4 Implementation of SQL+M

Working around limitation of sharing common subexpressions We observe that a straightforward application of common subexpression sharing from the MQO literature cannot by itself identify the memoization optimizations Perada uses in MBB/dom and CBB/streak. To see why, consider the following query (simplified and generalized) used to compute a perturbation (parameterized by *id*) and test whether it is an answer for MBB/dom and CBB/streak:

```
SELECT COUNT(*)
FROM R r1, R r2
WHERE r1.key = id
AND r2.x >= r1.x AND r2.y >= r1.y
AND (r2.x > r1.x OR r2.y > r1.y)
HAVING COUNT(*) <= κ;
```

Given multiple instances of the above query with different *id* values, MQO can identify the inequality self-join between `R` and itself as a common subexpression. Let us suppose MQO also recognizes that counts can be obtained by grouping. The combined query would be (where `Params` contains the set of *id*s associated with the query instances):

```
SELECT r1.key, COUNT(*)
FROM R r1, R r2, Params
WHERE r1.key = Params.id
AND r2.x >= r1.x AND r2.y >= r1.y
AND (r2.x > r1.x OR r2.y > r1.y)
GROUP BY r1.key HAVING COUNT(*) <= κ;
```

At this point, the database optimizer would take over and optimize this combined query. Unfortunately, it will likely not realize that

memoization of counts by $(\texttt{x}, \texttt{y})$ can be a good execution strategy. In other words, a straightforward application of common subexpression sharing across multiple query instance can fail to identify memoization opportunities.

We decided to give SQL+M a little boost, using the following observation: if we had used $(\texttt{x}, \texttt{y})$ as query parameters instead of *id*, then MQO could recognize different query instances with identical $(\texttt{x}, \texttt{y})$ pairs as common subexpressions, thereby enabling memoization. In effect, we manually "pre-condition" the query instances to workaround MQO's limitation. The resulting combined query (ignoring `Params` for simplicity) becomes:

```
WITH XY(x, y) AS (SELECT DISTINCT x, y FROM R),
  XYC(x, y, c) AS
    (SELECT x, y, COUNT(*) FROM R, XY
     WHERE R.x >= XY.x AND R.y >= XY.y
     AND (R.x > XY.x OR R.y > XY.y)
     GROUP BY x, y HAVING COUNT(*) <= κ)
SELECT * FROM XYC NATURAL JOIN R;
```

The definition of `XY` effectively hints at the memoization opportunity. Note that doing so demands no less developer expertise than having to identifying this memoization optimization to Perada.

A deeper question is why we cannot teach a database optimizer to pick up memoization automatically for the second query above. We believe that we can, and as mentioned in Section 8, we are actively working on techniques to derive such optimizations automatically using static query analysis. The results could benefit database query optimization in general, not just perturbation analysis or MQO.

Applying multiple sliding-window aggregates For Congress/vote and CBB/streak (step 1), perturbations conceptually involve computing an aggregate function (`AVG` for Congress/vote and `MAX` for CBB/streak) over a time series for different time periods. Although all periods are distinct, many have considerable overlap, presenting opportunities for shared processing. To implement SQL+M, we adapted ideas for processing multiple sliding-window aggregates from [3].

For `AVG` in Congress/vote, we implemented the algorithm for "subtractable" aggregate functions from [3], which is essentially the same as the basic $O(L+\ell^2)$ algorithm described in Appendix A.3.

For `MIN` in CBB/streak (step 1), we implemented the *B-INT* algorithm from [3] for distributive aggregate functions. Given a time series, we can precompute the aggregate function f over a hierarchy of *base-intervals* divided into levels; each base-interval is the disjoint union of multiple consecutive base-intervals in the lower level (Figure 10). In our setting of CBB/streak, the base-intervals at the bottom level correspond to individual games (streaks of length one). Precomputation is performed in a bottom-up fashion: as we move up the levels, we further aggregate the result from lower levels. By storing the base-intervals and the associated aggregate results in a search tree, the aggregate over any query interval can be computed from a logarithmic number of disjoint base-intervals—at most two on each level. (B-INT also handles updates, but that aspect is not relevant to our application.)

There are numerous techniques MQO that we could have implemented, but we felt the above two are a natural fit. In particular, [3] itself provides additional algorithms that handle filters on top of window aggregates with different filter thresholds and window sizes; we did not consider them as they are only approximate for `SUM` and `COUNT`. The landmark-based algorithms from [3] are also less applicable in our setting, because they target periods that are clustered in particular ways. *Predicate indexing* [4] is another approach that works by grouping many range filters together using a single predicate index, but it operates at the tuple level (returning all filters that a tuple satisfies) and does not by itself handle aggregation as the techniques we chose to implement.

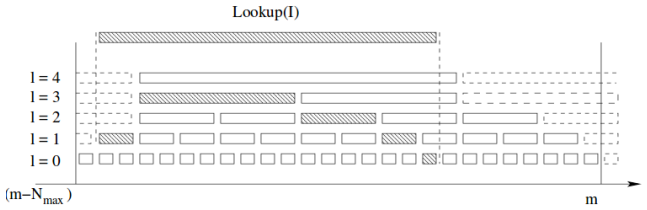

Figure 10: Illustration of the hierarchical structure used by the B-INT algorithm (Figure 2 of [3]).