FunctionGuard: A Query Engine For Expensive Scientific Functions In Relational Databases

Anh Pham, and Mohamed Eltabakh
Worcester Polytechnic Institute (WPI), MA, USA.
Computer Science Department
{*atpham, meltabakh*}@*cs.wpi.edu*

Keywords: Caching Techniques, Expensive User-Defined Functions, Scientific Applications, Query Processing.

Abstract: Expensive user-defined functions impose unique challenges to database management systems at query time. This is mostly due to the black-box nature of these functions, the in-ability to optimize their internals, and the potential inefficiency of the common optimization heuristics, e.g., "selection-push-down'. Moreover, the increasing diversity of modern scientific applications that depend on DBMSs and, at the same time, extensively use expensive UDFs is mandating the design and development of efficient techniques to support these expensive functions. In this paper, we propose the *"FunctionGuard"* system that leverages disk-based persistent caching in novel ways to achieve across-queries optimizations for expensive UDFs. The unique features of FunctionGuard include: (1) Dynamic extraction of dependencies between the UDFs and the data sources and identifying the potential cacheable functions, (2) Cache-aware query optimization through newly introduced query operators, (3) Proactive cache refreshing that partially migrates the cost of the expensive calls from the query time to the idle and under-utilized times, and (4) Integration with the state-of-art techniques that generate efficient query plans under the presence of expensive functions. The system is implemented within PostgreSQL DBMS, and the results show the effectiveness of the proposed algorithms and optimizations.

1 Introduction

Database systems provide an eminent support to scientific applications in various domains including biology, healthcare, astronomy, ornithology, among others. Many of these applications utilize DBMSs for managing large-scale datasets, e.g., biological databases in Genobase (http://ecoli.naist.jp/GB8/), Ecoli-House (http://www.porteco.org/), Ensemble project (http://www.ensembl.org/), and UniProt database system (http://www.ebi.ac.uk/uniprot), and healthcare databases in H.CUP (https://www.hcup-us.ahrq.gov/) and HealthCatalyst (http://www.healthcatalyst.com/). All of these databases leverage the desirable characteristics of DBMSs including advanced query optimization, indexing techniques, concurrency and recovery control, and ensured consistency.

One of the challenging requirements in scientific applications is that the targeted analysis usually goes beyond the standard SQL operations, e.g., selection, projection, join, and grouping and aggregation, to more complex analysis functions such as sequence alignment, prediction functions, quality assessment tools, image processing, feature extraction and dimensionality reduction, etc. These functionalities when integrated inside the DBMS they are typically modeled as stored procedures or functions, and hence they are treated as black box operations with very limited optimizations. Most modern DBMSs support implementing the user-defined functions in external languages, e.g., C or Java. Since these functions are usually expensive, they impose high overhead—and sometimes a bottleneck—to query execution.

Optimizing such expensive functions has been addressed in previous work (Hellerstein and Stonebraker, 1993; Hellerstein and Stonebraker, 1993; Chaudhuri and Shim, 1996; Scheufele and Moerkotte, 1998; Scheufele and Moerkotte, 1998). Most of these optimization techniques relay on estimating the cost of these functions, and then generating query plans that try to minimize the invocation of expensive functions. For example, if a selection predicate invokes an expensive UDF, then it can be more efficient not to use the optimization rule "selection-before-join", and instead postpose checking the expensive selection predicate until after the join operation. Other techniques use main-memory caching in which the invocations within a given single query are cached in main memory, and then if the same input parame-

ters are used multiple times during the execution, then the cached results are used to avoid redundant invocations (Hellerstein and Naughton, 1996).

However, as scientific functions are getting more complex and storage is getting less expensive, an attractive—and yet unexplored—approach is to use larger and persistent disk-based caches to maintain the output from these expensive functions for longer time. The advantages of this approach include: (1) Unlike main-memory caches which optimize a single query, persistent caches can optimize multiple queries, and (2) As the data size is getting larger, persistent caches will offer much larger space to store more data, and hence increase the cache hit ratio and avoid more invocations to these functions.

In this paper, we propose a new mechanism for optimizing expensive user-defined functions in relational database systems. We propose the *FunctionGuard* system that leverages disk-based persistent caching to achieve across-queries optimizations supported by proactive cost-based cache refreshing techniques and cache-aware query optimization. FunctionGuard relays on the following novel features: **(1) Dynamic Extraction of Dependencies:** We develop a function analysis tool that automatically extracts the dependencies of the user-defined functions from their source codes. These dependencies may include the input arguments passed to the function, access to database tables, external files, wall clock, and invocation to other functions. The output from this process is a complete functional dependency graph that captures all elements that a given function F depends on. **(2) Identifying Cacheable Functions and Caching Results:** Based on the dependency graph, the system identifies which functions are *cacheable*, i.e., their outputs can be temporarily cached and reused across multiple invocations. A cacheable function must have all its dependencies (immediate children in the dependency graph) identified as *trackable*, i.e., the DBMS can track whether or not these dependencies have changed from the last invocation. For example, an access to a local DB table is a trackable dependency, whereas an access to an external DB table is not. For cacheable functions, their output results are cached for subsequent re-use. **(3) Cache-Aware Query Optimization:** We extend the query engine of FunctionGuard to take into account the cached results of expensive UDFs. The caching may partially or completely eliminate the need for the function invocation at query time. We propose new query operators and algorithms for efficiently integrating the cached results in the query pipeline. We also propose cache replacement mechanisms that are suitable for disk-based caches in contrast to main-memory caches. **(4) Proactive Execution and Pre-fetching of Results:** For a cacheable function F, when one of F's trackable dependencies change, the output cache will be invalidated. One possible approach is to wait for the next invocation of F to refresh its cache. In contrast, FunctionGuard deploys a cost-based proactive mechanism in which the system may proactively refresh F's cache output using the most recently or frequently used arguments. In that case, the execution overhead of the function is *partially* shifted to the times when the system is idle or under utilized in contrast to the query time.

FunctionGuard is not a replacement for the existing techniques, instead it is complementary to all algorithms that generate efficient query plans in the presence of expensive UDFs, e.g., (Hellerstein, 1994; Hellerstein and Stonebraker, 1993; Chaudhuri and Shim, 1996; Scheufele and Moerkotte, 1998; Scheufele and Moerkotte, 1998). That is, FunctionGuard takes as input a query plan generated from any of these systems, and then it updates the plan by replacing the operator used for function invocations by a more complex cache-aware operators that utilize the available cache.

The rest of the paper is organized as follows: In Section 2, we present the related work. In Section 3, we present the proposed mechanisms for building the dependency graph and managing the cache. The cache-aware query processing, and proactive cache update are presented in Sections 4, and 5, respectively. Finally, the experimental evaluation and conclusion remarks are included in Sections 6, and 7, respectively.

2 Related Work

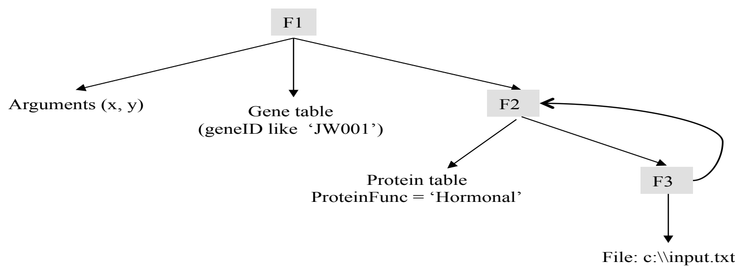
Scientific algorithms and techniques usually require super-linear or even N^2 or N^3 time complexity to process N data points (Gray et al., 2005; Haas et al., 2001). These expensive functions when implemented inside the DBMSs represent a major optimization challenge at query time. Several techniques have been proposed to optimize the execution of these expensive functions, e.g., predicates that involve expensive functions or expensive subqueries (Chaudhuri et al., 2002; Chaudhuri and Shim, 1993; Hellerstein and Stonebraker, 1993; Munagala et al., 2007). The work in (Chaudhuri et al., 2002) has explored the usage of data mining models to derive query predicates, and then how the predicates can be used for efficient execution. In contrast, the techniques presented in (Hellerstein, 1994; Hellerstein and Stonebraker, 1993; Chaudhuri and Shim, 1996; Scheufele and Moerkotte, 1998; Scheufele and Moerkotte, 1998) have all focused on optimizing the query plan by not following the traditional heuristic of "predicate-pushdown" since the predicates in the case of expen-

```
Create Function F1(int x, varchar y)
{
    ....
    Select gene_name
        From Gene
        Where geneID like 'JW00%' ;
    ....
    Execute F2();
}
```

```
Create Function F2()
{
    Select protein_name
        From Protein
        Where proteinFunc = 'Hormonal' ;
    ....
    IF (condition)
        Execute F3()
}
```

```
Create Function F3()
{
    ....
    File *f = fopen("c:\\input.txt","r");
    ....
    Execute F2();
}
```

(a) Expensive functions in the database.



(b) Dependency Graph between functions and data sources.

Figure 1: Examples of expensive UDFs in scientific applications and their dependency graph.

sive functions can be even more expensive than a join operation. Thus, these techniques proposed new cost models and extensions to existing query optimization techniques to find more efficient query plans.

Capturing the semantics of the external functions has been addressed in (Chaudhuri and Shim, 1993; Chaudhuri and Shim, 1996). Users provide semantics information and cost estimates for their UDFs, and these information is integrated with the query optimizer to choose the cheapest query plan. The presence of expensive functions in queries has been addressed in different contexts including multimedia repositories (Chaudhuri and Gravano, 1996; Zhang et al., 2012), object-relational databases (Hellerstein, 1998), and continuous queries (Munagala et al., 2007; Denny and Franklin, 2006). For example, The work in (Denny and Franklin, 2006) has proposed approximation techniques to run the expensive UDFs to only the accuracy needed by the query instead of consuming the entire input. The algorithm presented in (Chang and Hwang, 2002) proposes a strategy to answer top-k ranking queries with minimal number of probing to evaluate objects, and hence minimizing the invocation of the involved expensive functions.

Although most of the existing techniques focus on query plan generation and optimal operator placement for evaluating expensive functions, few techniques have leveraged caching as a mean for optimization. The work in (Hellerstein and Naughton, 1996) has proposed a *hybrid main-memory cache* technique to avoid unnecessary and redundant invocations of expensive functions. Since the technique uses a main-memory cache it only optimizes a single query and does not span across queries.

The FunctionGuard system is distinct from the previous techniques in that: (1) It leverages disk-based caching techniques, and hence the cached data will optimize multiple queries and the amount of cached entries can be more scalable compared to main-memory caches, (2) The system does not search for an optimal query plan, and hence it is complementary to existing techniques, i.e., FunctionGuard can be used on top the existing techniques, e.g., (Hellerstein, 1994; Hellerstein and Stonebraker, 1993; Chaudhuri and Shim, 1996; Scheufele and Moerkotte, 1998), as will be explained in Section 4. And (3) FunctionGuard includes novel mechanisms for disk-based cache maintenance, which are different from those developed for main-memory caches.

3 Dependency Graph and Results Caching

3.1 Dependency Graph

Maintaining the dependencies among the user-defined functions and the data sources, e.g., database tables, flat file, wall clock, or any other sources of data, is an essential metadata information in the system. This is because these dependencies will enable the system to keep track of whether or not a specific cache (a materialized output from a function) is still valid. The cache of a given function F is considered valid as long as all elements on which F depends do not change.

FunctionGuard constructs a functional dependency graph G based on the functions' definitions (See Figure 1). The graph $G = (V, E)$ is a directed graph, where the nodes V represent either a UDF or a data source. Each non-leaf node represents a UDF and each leaf node represents a data source. Each

Source type	Additional Info	Trackable
Database table	Predicates in the form of "colum <op> constant"	Yes
External file	---	Yes
Wall Clock	---	No
Random Generation	---	No
Other sources	---	No
Func Calls	---	Yes (if cacheable)

Figure 2: Types of Data Sources Used in UDFs.

edge $e = (v_1, v_2) \in E$ represents a dependency from node v_1 (must be a UDF) to node v_2 (either another UDF or a data source), which indicates that v_1 depends on or accesses v_2. For example, the functions illustrated in Figure 1(a) will map to the dependency graph in Figure 1(b). Function F_1 depends on its two input arguments x and y, the `Gene` table in the database, and function F_2. Therefore, if, for example, the data in table `Gene` have changed, then the cached output of F_1 may be invalidated. After extracting the dependencies of each of the three functions, the produced graph will be as depicted in Figure 1(b).

It is worth noting that the graph captures coarse grained dependencies among the UDFs and the data sources. This means that a function F depends on a data source s if there is any execution path of F that accesses s. And thus, these dependencies are static based on F's definition and independent from the input arguments passed to F. For example, function F_2 depends on F_3 only under a certain condition. However, the dependency graph does not capture such fine-grained dependency because it is very expensive to capture the dependencies for each execution path especially is complex functions. The graph may also contain cycles, e.g., the cycle between functions F_2 and F_3 in Figure 1(b). These cycles do not necessarily mean that there are cycles at execution time because F_2 may be referencing F_3 in specific execution paths or under certain conditions as shown in the figure.

Graph Construction: To construct the graph, we built a tool that takes a function's source code in C, Java, or PL/SQL, and returns its dependencies. Hence, given a set of UDFs in the database, the tool incrementally builds the dependency graph. For a given function F, the types of the extracted data sources along with their properties are summarized in Figure 2. The tool extracts references to database tables, and for each table T it captures any predicates on T in the form of *"columnName <op> constant"*. These predicates are important as they will help narrowing down the false-positive decisions of invalidating a cache of a function F although the actual change to a database table is irrelevant to F's output. The database tables are defined as *trackable* data sources since FunctionGuard can track whether or not it has changed from the last invocation. The system also extracts any access to flat files in the file system, and defines them as *trackable* data sources since FunctionGuard can track whether or not a file has changed by checking its *last modification* timestamp. References to the wall clock, random generation, or other sources, e.g., references to external databases, are also extracted and defined as *non-trackable*. Finally, invocations to other functions are also extracted as illustrated in Figure 2.

Based on the extracted dependencies, a function F is categorized as *cacheable* iff all its dependent data sources are *trackable* and all its dependent functions are *cacheable*. Otherwise, F is categorized as *non-cacheable*. In the presence of a cycle in the dependency graph, if none of the functions involved in the cycle is categorized as *non-cacheable*, then all of them will be categorized as *cacheable*. For example, functions F_2 and F_3 in Figure 1(b) will be both categorized as *cacheable*.

3.2 Cache Management

As an initial setup, the database admin needs to allocate a certain disk space M in the database for the cache managed by FunctionGuard. Initially, FunctionGuard will divide this space equally between the expensive functions. However overtime, the system will monitor the usage of these expensive functions and dynamically assign priorities to them, which will reflect on the amount of cache allocated to each function. The priorities are computed as follows:

Priorities of Expensive Functions: Assume that there are N expensive functions in the system, denoted as $\mathcal{F} = \{F_1, F_2, ..., F_N\}$. Our goal is to make the system adaptive to the most recent usage of the expensive functions and not highly dependent on the old history. In order to achieve this, the system maintains a catalog table that stores for each function F_i its usage over the last K queries executed in the system. The usage of function F_i in query Q_j includes two parameters: $F_{ij}.Frequency$ and $F_{ij}.TotalTime$, where the former captures the number of times F_i is invoked by Q_j, and the latter captures the total time consumed by F_i from all its invocation by Q_j. As more queries are executed by the system, each new query Q_{new} referencing function F_i will replace the oldest entry in F_i with a new entry for Q_{new}. Moreover, for each function $F' \in \mathcal{F}$ that is not referenced by Q_{new}, the oldest entry will be deleted and a new entry with zero parameters is added to F' history. The mechanism ensures that the system will adapt to the most recent K queries in the system. Given these parameters, the priority of each function F_i is computed as follows:

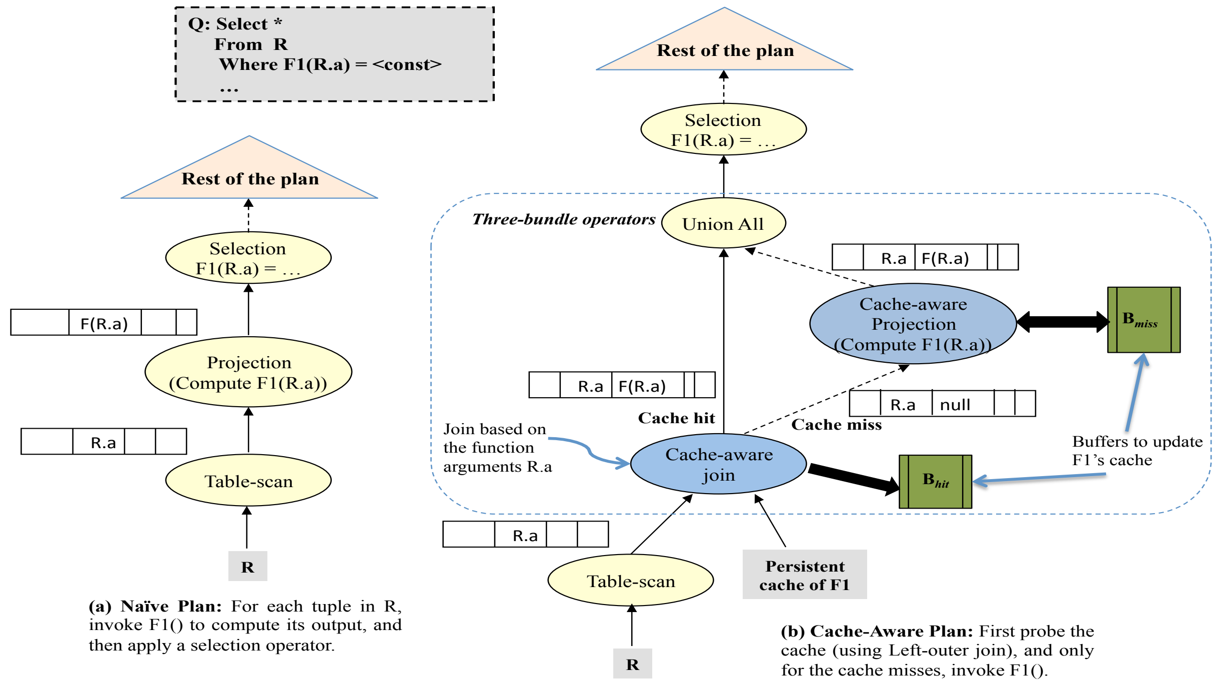


Figure 3: Example of Cache-Aware Query Plans in FunctionGuard.

$$F_i.FreqScore = \sum_{\forall\, j \in K} F_{ij}.Frequency \;/\; SumFreq$$

$$F_i.TimeScore = \sum_{\forall\, j \in K} F_{ij}.TotalTime \;/\; SumTime$$

$$F_i.priority = \tfrac{1}{2}(F_i.FreqScore \;+\; F_i.TimeScore)$$

where, *SumFreq* (and *SumTime*) is the sum of all functions' frequencies (and times) over all K queries. As a result, each of $F_i.FreqScore$, $F_i.TimeScore$, and ultimately $F_i.priority$ should be between [0,1] interval with total sum of each measure as 1. The $F_i.priority$ will then be used as the percentage of the cache allocated to function F_i.

Cache Maintenance: When a function F_i is invoked at query time, the system needs to perform two main tasks: (1) Identify which cached values for F_i can be re-used, and (2) update the F_i's cache by either adding more new entries to the cache (if space permits), or by replacing less important entries with more important entries. Since the maintained caches in FunctionGuard are disk-based, their manipulation, e.g., probing and replacement will be different from main-memory caches. In this section, we describe the structure of a cache entry and how to measure its importance (in the case of replacement), then in Section 4 we present how the cache is efficiently manipulated at query time.

For a given function F_i, each entry in the cache will have three components: **(1)** The input arguments used when invoking F_i, **(2)** The output results, and **(3)** A frequency capturing how many times F_i has been called with these input arguments. An entry with a larger frequency value means that the corresponding input arguments are more frequent, and hence this entry gets more weight and tend to stay longer in the cache. More formally, if the cache allotted for F_i holds at most $Count_{F_i}$ entries, then these entries will be the ones with the highest frequency values.

4 Cache-Aware Query Processing

If a user's query involves an expensive function, then FunctionGuard will utilize the available caches to avoid unnecessary invocations of these functions. By doing so, the execution and response times of a given query can be significantly enhanced—the experiments in Section 6 illustrate that the speedup can be more than two orders of magnitude. In this section, we introduce the cache-aware query processing operators, how the persistent cache is utilized at query time, and how the cache is maintained and updated.

4.1 Bundled Cache-Aware Operators

Considering the query in Figure 3 that involves an invocation to an expensive function F_1 in the *Where* clause. Assume that the query plan depicted in Figure 3(a) is the plan generated from one of the state-of-art techniques. In this plan, each input tuple from relation R will go through a *projection* operator to compute F_1 over attribute $R.a$. And then, a *selection* operator will pass the tuples satisfying the predicate(s) to

the rest of the query plan. Therefore, in this plan, the dominant factor in the execution time will be *(cost(F_1) x |R|)*, where *cost(F_1)* is the cost for a single invocation of F_1 and $|R|$ is the cardinality of R.

In FunctionGuard the query plan will be modified by replacing the *projection* operator that invokes F_1 by a *three-bundle operators*, which consist of a newly introduced *cache-aware join* operator, a *cache-aware projection* operator, and a *union all* operator (See Figure 3(b)). The two query plans presented in Figure 3 are equivalent plans, i.e., the produce the same results given the same input. The *three-bundle operators* can be used in replacement for the standard *projection* operator regardless of whether the function invocation is in the projection list, *Where* or *Having* clauses. The *cache-aware join* operator will join the input tuples with the persistent cache of function F_1 based on the input parameters passed to the function, e.g., *R.a* in our example. If there is a cache hit, then the output will be forwarded to the *union* operator. Otherwise, the tuples are forwarded to the *cache-aware projection* operator as illustrated in the figure to compute F_1 on these tuples. For each invocation of F_1 from the operator, the output value is cached in a main-memory buffer, termed B_{miss}. The usage of this buffer is twofold. First, subsequent inputs to the *cache-aware projection* operator will be checked first against the entries in B_{miss} buffer—which acts as a second-level cache. If an entry is found, then the invocation to F_1 is skipped. Second, this buffer along with the B_{hit} buffer will be used to update F_1's persistent cache.

To update F_1's cache, we need to reflect the usage of the current query on that cache, e.g., how many times the function is called, each input argument is used how many times, and the total execution time consumed by F_1. Therefore, the proposed cache-aware operators maintain main-memory buffers in order to collect these information (Refer to Figure 3(b)). In Figure 4, we present the algorithm that the *cache-aware join* and *cache-aware projection* operators use to update the B_{hit} and B_{miss} buffers, respectively. For the B_{hit} buffer, when the *cache-aware* join operator finds a cache hit in F_1's persistent cache, an entry is added to B_{hit} capturing the function's input arguments having a frequency set to 1. Any subsequent appearance of the same input arguments will only increment the corresponding frequency. For the cache misses from the join operators, the *cache-aware projection* will invoke $F1$ to compute the functions output, and an entry consisting of $(r.a, F_1(r.a), 1)$ will be added to the B_{miss} buffer. Any subsequent appearance for the same *r.a* will be served from the B_{miss} buffer without invoking F_1 as illustrated in Figure 4, and will only increment the corresponding frequency to keep track of how many times the same input argument is used.

Memory Constraints for Buffers: it is possible

Managing buffers in the three-bundle operators

Assumptions
- The expensive function is *F1*
- The input argument to *F1* is *r.a*, where *r* is a data tuple and *a* is a data attribute

B_{hit} Buffer // each entry is: (argument list, frequency)
- For each input tuple r to the cache-aware *join* operator
  - If (cache hit)
    - If (an entry for *r.a* exists in B_{hit}) Then
      - Increment its frequency value
    - Else
      - Add an entry for r.a with frequency= 1
    - End If
  - End If

B_{miss} Buffer // each entry is: (argument list, output value, frequency)
- For each input tuple *r* to the cache-aware *projection* operator
  - If (an entry for *r.a* exists in B_{miss}) Then
    - Retrieve its output value to update *r*
    - increment the frequency for this entry
  - Else
    - Execute function *F1*
    - Add an entry in B_{miss} (*r.a*, *F1* output, 1)
  - End If

Figure 4: Managing Buffers in the Three-Bundle Operators.

that the B_{hit} and B_{miss} buffers get filled up and cannot take more tuples. Therefore, both operators *cache-aware join* and *cache-aware projection* have strategies to spill these buffers to disk when needed. The case of the B_{hit} buffer is easier since the *cache-aware join* operator is only writing to this buffer without reading back from it. Hence, when it gets full, its entries are sorted based on the function's arguments, e.g., *r.a* in our example, and then written to disk. For the B_{miss} buffer the scenario is different since the projection operator is writing and reading from it. Therefore, if this buffer is full, then the *cache-aware projection* operator will write any input tuple *r* that does not have a successful cache hit in B_{miss} to disk without invoking the F_1 function. Therefore, when all tuples are consumed, it will be guaranteed at that time that no more tuples can have a cache hit with the current content of B_{miss}. Hence, the buffer will be sorted based on the function's arguments, and then written to disk. After that, the data tuples spilled to disk are read and processed one at a time and by invoking the F_1 function over them. And the B_{miss} buffer gets populated again with the function's outputs until all tuples are processed. By deploying these strategies, then the number of invocations to the expensive function F_1 is guaranteed to be minimal—The number of invocation is zero for arguments that exist in the persistent cache, and it is one for the other argument values.

Special Case−No Persistent Cache: A special case to the algorithm presented above is that if the expensive function has no persistent cache. This may happen if it is the first time a given function is used

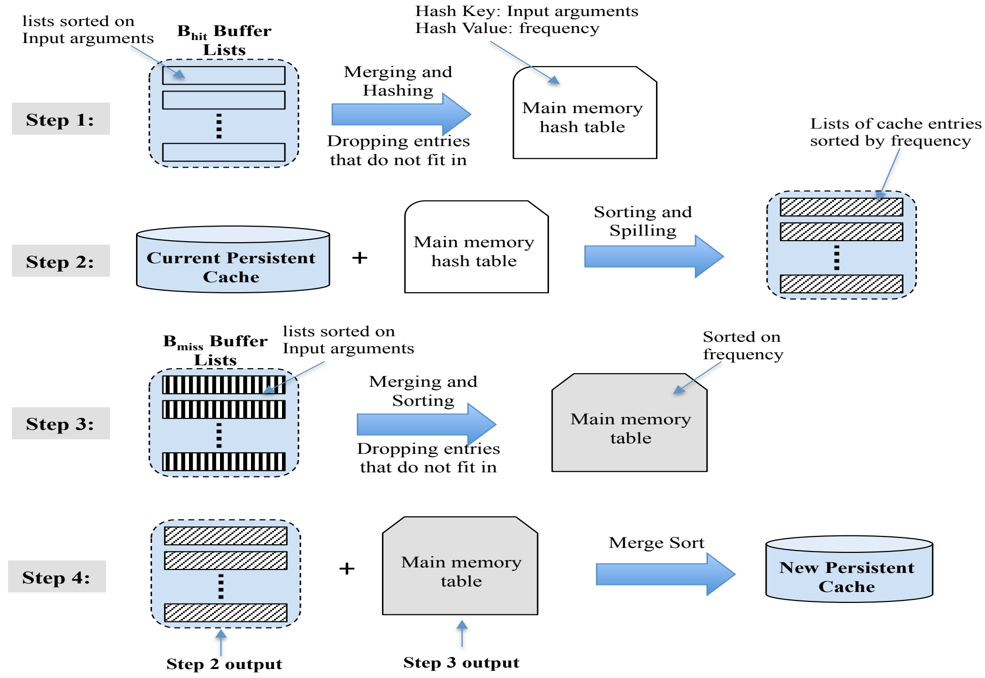


Figure 5: Updating a Persistent Cache of an Expensive Function.

within a query, or if the existing cache is invalidated due to a change in one of the data sources on which the function depends. This information is maintained in a system catalog table as will be presented in Section 5. In either case, FunctionGuard will not use the *three-bundle operators* since there is no persistent cache to use. Instead, the system will only use the *cache-aware projection* operator to evaluate the expensive function with each input tuple, caches the results in the B_{miss} buffer, and then the buffer will be written to the persistent cache after the query finishes.

4.2 Persistent Cache Update

After the query is completed, the information collected in the B_{hit} and B_{miss} buffers will be used to update the function's persistent cache on disk, i.e., the current most important entries should be kept in the cache. However, since the cache is persistent on disk, it is more complicated and expensive to update the cache compared to main-memory caches. Thus, in FunctionGuard the cache of a given expensive function F is not updated after each query, instead it is updated after β queries referencing F—β is a configuration parameter in the system that can be defined per function. As a result, the cost of a cache update is amortized over the β queries.

The procedure for refreshing and updating the cache of a function F involves four main steps as illustrated in Figure 5. The overall goal from this procedure is to revisit the cache entries in the persistent cache as well as the B_{hit} and B_{miss} buffers (at that time these buffers are written to disk from the last β queries), and re-populate the cache to keep the entries that are referenced more frequently, i.e., the *frequency* value is high. The procedure works as follows. In Step 1, the algorithm reads all B_{hit} lists that are written to disk in parallel. Since each list is sorted based on the input arguments, then using a multi-way merge, we can efficiently merge all occurrences of a given input arguments I. The output is stored in a main-memory hash table, where the hash key is the input argument I. The hash table does not need to maintain all the entries—which can be large—, we only maintain the entries having the largest frequency that can fit the hash table. For the other entires, we assume their frequency is small and we approximate them to zero. The advantage of having the hash table fits in the main memory will be clear in Step 2 of the update procedure. In Step 2 (See Figure 5), the algorithm scans the current persistent cache for function F, and for each entry e =*(Input arguments, output, frequency)*, the hash table built in Step 1 is probed (using the input arguments values) to update the frequency of this entry. Since the hash tables fits in memory, then we only need one scan over the persistent cache to complete this step. The output from this step is stored in a memory buffer, and when it gets full, it will be sorted based on the frequency values and then spilled to disk as depicted in the figure.

Step 3 is similar to Step 1 in which the algorithm will merge the entires in the B_{miss} lists to get the to-

tal frequency for each input arguments and produce the output entry. The main difference is that the output table will not be a hash table, instead it will be a main memory table sorted based on the frequency values. This difference is due to the fact that the entries in the B_{miss} lists are guaranteed not to exist in the current persistent cache. In this step, we also use the same approximation strategy as in Step 1, where the memory table may not hold all entires from the B_{miss} lists, and in that case, we keep only the entires that have the largest frequencies and approximate the others to zero. The final step involves one scan over the lists produced from Step 2 (sorted based on their frequency values) and applying a merge sort algorithm over these lists and the output from Step 3 (also sorted based on the frequency values). The output from this step is the new persistent cache.

```
Create Trigger FunctionGuard_I
After Insert On T
For each raw
Begin
        // ***F1 dependency predicates section***
        - If (``new'' vector satisfies predicates p1) Then
              - Insert into the catalog table values
                        ("T", "Insertion", "F1", timestamp)
        - End If;
        - If (``new'' vector satisfies predicates p2) Then
              - Insert into the catalog table values
                        ("T", "Insertion", "F1", timestamp)
        - End If;

        // ***F2 dependency predicates section***
        - If (``new'' vector satisfies predicates p3) Then
              - Insert into the catalog table values
                        ("T", "Insertion", "F2" , timestamp)
        - End If;
End;
```

Figure 6: Example of DB triggers that monitor cache invalidation on data source T.

5 Invalidation and Proactive Cache Refreshing

The cached output from an expensive function F remains valid as long as the data sources on which F depends remain unchanged. These data sources and dependencies are captured in the dependency graph maintained by the system (Section 3). The system keeps track of any changes to the data sources in different ways depending on the source's type as follows:

• **External Files:** FunctionGuard tracks the changes in the external files through the operating system by checking their last modification timestamps. However, there is no generic way on top of the OS that detects instantaneously a change in the file system's information. Nevertheless developing a periodic job that retrieves the last modification time of the files will not guarantee a timely detection of these changes. Therefore, FunctionGuard performs this check at query time, i.e., given a query Q that invokes functions $F_1, F_2, ..., F_N$, the system will check the dependency graph to find which external files that these functions depend on, and then it decides on which functions will have their caches invalidates (if any).

• **Database Tables:** If a function F depends on a database table T, the system extracts any predicates used inside F on table T that limit the scope of T's tuples accessed by F. These predicates are in the form of *"columnName <op> constant"*. When this information is extracted, FunctionGuard automatically creates database triggers on T that monitor the insertion, deletion, or update of any data tuples satisfying the extracted predicates. These triggers insert records into the catalog table whenever a change in T that invalidates F's cache takes place. With large number of functions, the number of triggers created on a given table can be large, which may degrade the performance of any operation on that table (Hanson et al., 1999). Therefore, FunctionGuard does not create separate triggers, instead it creates one (or few) triggers and it augments more code to the trigger's body with each addition of a new expensive function. The pseudocode presented in Figure 6 give an example of such system-generated triggers. In the example, two expensive functions F_1 and F_2 depend on the database table T. Assume Function F_1 has two separate *select* statements; one concerns the tuples satisfying predicate p_1, and the other concerns the tuples satisfying predicates p_2. In contrast, Function F_2 contains a *select* statement having predicate p_3 on table T. Thus, one of the generated triggers on T is the *After Insert* trigger containing code as depicted in the figure. For example, any inserted tuple that satisfies either of predicates p_1 or p_2, will insert a tuple in the catalog table indicating that an *"insertion"* operation in Table *"T"* at timestamp *"timestamp"* invalidates the cache of function "F_1". This catalog table is checked by the system to decide whether or not a specific cache is invalidated.

Proactive Cache Refreshing: Refreshing and updating a cache is typically performed in a *passive* mode, i.e., when the system detects that a cache is invalid, all its entries will be deleted and the cache will be populated later by the next executed query. The FunctionGuard system deploys a more proactive feature that can save significant time at query execution. This feature works as follows. When the system detects that a change in a given database table will invalidate the cache of a given file F, a set of

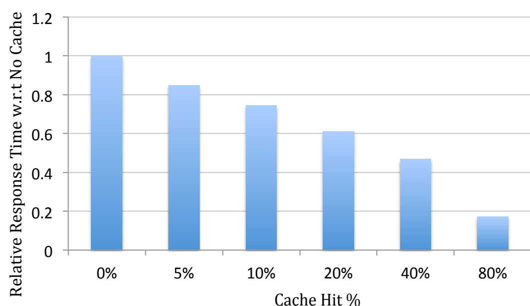


Figure 7: Effect of Caching on Performance.

the input arguments stored in the cache having the highest frequency will be extracted before deleting the cache. These input arguments represent the arguments most frequently used when invoking function F, and thus they will be stored in a log table. The database admins can then schedule the invocation of F using these logged entires to proactively populate F's cache. Such type of tasks can be scheduled at the low-utilization time of the system, and hence the overhead of executing the expensive functions will be partially shifted from the query time to the times when the system in under utilized.

There is always a tradeoff between the number of input arguments used in the proactive cache refreshing and overhead paid to perform the proactive execution. In the current version of FunctionGuard , the proactive execution is configured by the database admins for each expensive function. That is, for a given function F, there is a configuration parameters that specifies whether or not the *proactive execution* feature is ON or OFF. If ON, then there is another configuration parameters that specifies the percentage of input arguments to use in the proactive execution—By default it is set to 10%. In general, the proactive cache refreshing feature is a *best effort* feature, which means that if the system could not perform the proactive execution for all or part of the scheduled invocations, e.g., no idle or low-utilization time has been observed before the next query, then the un-executed calls will be discarded.

6 Experimental Evaluation

The experiments are conducted on a Dell OptiPlex 990 Desktop machine having Intel dual core i5 2400 Processor (3.1GHz, 6M), 4GB DDR3 memory, and 250GB SATA hard drive. We used real-world biological datasets from the UniProt database system (http://www.ebi.ac.uk/uniprot), more specifically the `Protein` table that consists of 200,000 protein tuples occupying around 600MBs in the database. For the expensive function, we used the BLAST (Basic Local Alignment Search Tool) program that enables comparing a given protein sequence with a library or database of other sequences. The program offers many functionalities, among which we used `blastp` function that searches protein sequences. The query we used is a simple select-project query from the `Protein` table that performs a sequential scan over the table,. The query contains one predicate over the protein sequence column that compares each sequence to the NCBI repository (Protein Data Bank).

The objective from the evaluation is to study: (1) The effect of caching on the queries' response time, (2) The cost involved in the cache update and maintenance, (3) The effect of the internal data structures, e.g., the buffers B_{hit} and B_{miss}, on the performance, and (4) The effectiveness of the proactive cache update on reducing the overhead at query time. Notice that we do not compare or study the performance of different query plans because, as we mentioned in Section 1, FunctionGuard uses the state-of-art techniques (Hellerstein, 1994; Hellerstein and Stonebraker, 1993; Chaudhuri and Shim, 1996; Scheufele and Moerkotte, 1998; Scheufele and Moerkotte, 1998) as a black box to generate an efficient plan. And hence, we focus on the cache-related performance.

In Figure 7, we study the effect of caching the results from the expensive predicate and then re-using it in the next query. The x-axis shows the cache hit ratio ranging from 0% to 80%, i.e., the ratio at which the expensive predicate is not executed because the results was cached. The y-axis shows the relative performance of FunctionGuard compared to having no cache. Since the expensive predicate is dominating the cost of the query, the performance is inversely proportional to the number of cache hits.

In Figure 8, we illustrate the overhead involved in updating the content of the cache−Recall that a cache should maintain only the entires with the highest priority as presented in Section 4.2. Since the cache is disk-persistent, the update cost can be expensive. The algorithm proposed in Section 4.2 amortizes the cost over multiple queries. The results depicted in Figure 8 show the total and amortized cost when varying the number of queries (β) between 1 and 16 (the x-axis). The figure indicates that as β increases, the cost will also increase. This is because the collected information from the multiple runs are accumulating, and hence it takes more time to reflect that on the cache. However, as illustrated in the figure, the increase is sub-linear and thus the amortized cost is reduced as β increases. Although higher β means less amortized cost, it has the drawback of delaying the update of the cache, and hence some important entries may not be reflected immediately until the next update. Therefore β is an important configuration parameter that

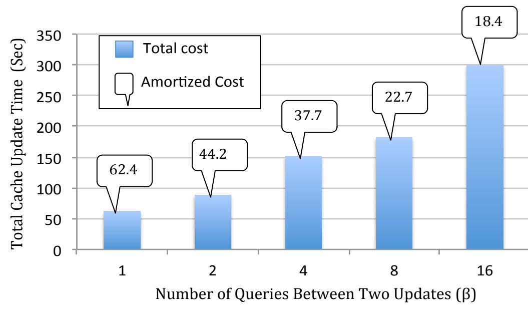


Figure 8: Cache Update Performance.

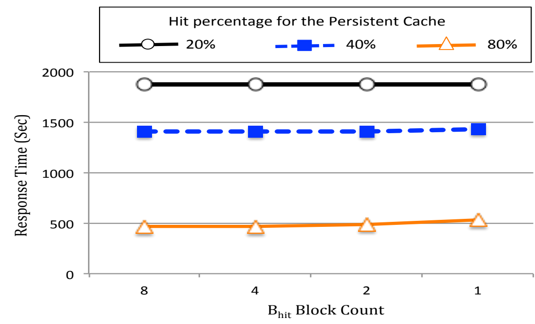


Figure 10: Effect of B_{hit} Block Size on Response Time.

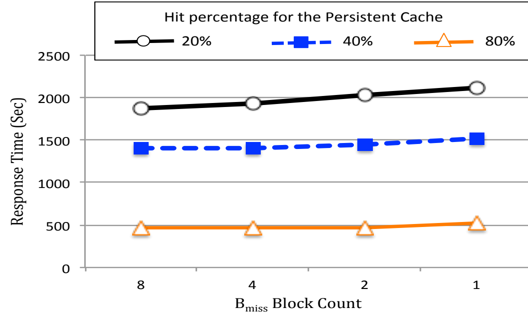


Figure 9: Effect of B_{miss} Block Size on Response Time.

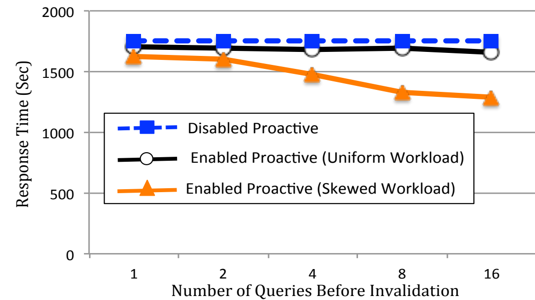


Figure 11: Proactive Cache Refreshing.

depends on the query rate, the system load, and the desired aggressiveness to update the cache. In the current version of FunctionGuard , β is set by the database admin, however, we plan to explore, as part of our future work, a cost model based on which β can be tuned dynamically.

The previous experiments reported in Figures 7 and 8 assume 8-Block main memory buffers for each of the B_{miss} and B_{hit} maintained by the *three-bundle operators*. Each block is of size 1MB. In the following experiments (Figures 9 and 10), we study the effect of the buffers' sizes on the query performance. We vary the number of blocks assigned to each buffer over the values from 8 to 1, and measure the query response time. For the B_{miss} buffer, as the size decreases, the number of tuples that encounter a cache miss in the B_{miss} buffer increases (See the *Cache-Aware Projection* operator in Figure 3). And as this number increases, the data tuples can be temporarily moved to disk until the first scan completes. As Figure 9 illustrates, for high percentage cache-hit for the persistent cache, e.g., 80%, most data tuples will be served by the *Cache-Aware Join* operator and they will skip the *Cache-Aware Projection* operator. In that case, the performance is less sensitive to the size of the B_{miss} buffer. In contrast, with lower percentage cache-hit for the persistent cache, e.g., 20%, most tuples will be forwarded to the *Cache-Aware Projection* operator. As a result, as the buffer size decreases, the query's response time increases.

For the B_{hit} buffer (Figure 10), the effect of the cache-hit percentage of the persistent cache is reversed. That is, as the hit percentage increases over 20%, 40%, to 80%, the *Cache-Aware Join* operator processes more tuples, and hence the utilization of the B_{hit} buffer will be higher. As illustrated in Figure 10, there is almost no effect on the performance in the case of 20% cache hit even if the size of the B_{hit} buffer is one memory block. However, the effect if clear when the cache hit is 80%. Comparing the results in the Figures 9 and 10, shows that tuning the size of B_{miss} is more important than B_{hit}. The reason is that the *Cache-Aware Join* operation is less expensive than buffering the data tuples (and probably writing them to disk) and the iterating over them again.

In Figure 11, we study the effect of the proactive cache refreshing when the cache is invalidated, e.g., because on of the dependent sources has changed. In the experiment, we manually controlled when to invalidate the cache, i.e., after 1, 2, 4, 8, or 16 user's queries (the x-axis in the figure). The effect of this is that the if the invalidation takes place after several queries, then FunctionGuard has a higher chance to learn more about the invocation pattern of the expensive function. Hence, the predicted and proactively cached entires are more likely to be used. To setup the experiment, we simulated a workload as follows. We divided the data tuples in the `Protein` table into 5 segments, each one consists of 40K tuples. In the first experiment (labeled *"Uniform Workload"* in Figure 11), all segments have the same probability of being involved in (touched by) a user's query. There-

fore, it is harder for the system to learn a useful invocation pattern. In contrast, in the second experiment (labeled *"Skewed Workload"* in Figure 11), the each segment has a different probability of being touched by a user's query. We set the probabilities to be exponentially decreasing, e.g, Segment s_i has double probability than Segment s_{i-1} of being queried. A given user query will touch 50,000 tuples across the 5 segments according to the probabilities set in each experiment. And the proactive mechanism is limited to proactively caching only 10,000 entires.

As the experiments show, the benefit from the proactive execution and quality of the predicted invocation pattern depends on the user's workload. In the case of *Uniform Workload*, all tuples have the same chances of being queried, higher number of the predicated and proactively cached values are wrong predictions. Therefore, the savings at query time is not large even if we increase the number of queries before a given invalidation. In contrast, the results in the case of *Skewed Workload* show larger savings because the system is able to identify the invocation patterns more likely to be used in the future. And hence, the cache hits get higher. This is more effective as the number of queries before the invalidation is relatively large, e.g., 8 or 16.

7 Conclusion

We proposed the FunctionGuard system for efficiently incorporating expensive functions in relational database queries. FunctionGuard is distinct from existing systems in that it leverages disk-based caches in novel ways to speedup query execution by avoiding unnecessary invocations. It addition, it can be integrated with any of the state-of-art techniques that build optimal query plans in the presence of expensive functions. The unique features of FunctionGuard include: (1) Automated mechanisms for analyzing expensive functions and building the corresponding dependency graph between functions and data sources, (2) Cache-aware query processing and optimizations based on the *three-bundle* operators to integrate the cached data into the query pipeline, And (3) mechanisms for updating and refreshing the disk-based caches in batch-optimized and proactive ways. The empirical evaluation demonstrated the effectiveness of the proposed system to speedup queries and enhance the utilization of the existing cache.

REFERENCES

Chang, K. C.-C. and Hwang, S.-w. (2002). Minimal probing: Supporting expensive predicates for top-k queries. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 346–357.

Chaudhuri, S. and Gravano, L. (1996). Optimizing queries over multimedia repositories. pages 91–102.

Chaudhuri, S., Narasayya, V., and Sarawagi, S. (2002). Efficient evaluation of queries with mining predicates. In *ICD*, pages 529–540.

Chaudhuri, S. and Shim, K. (1993). Query Optimization in the Presence of Foreign Functions. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, pages 529–542.

Chaudhuri, S. and Shim, K. (1996). Optimization of queries with user-defined predicates. In *ACM Transactions on Database Systems*, pages 87–98.

Denny, M. and Franklin, M. (2006). Operators for expensive functions in continuous queries. In *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on*, pages 147–147.

Gray, J., Liu, D. T., Nieto-Santisteban, M., Szalay, A., DeWitt, D. J., and Heber, G. (2005). Scientific Data Management in the Coming Decade. *SIGMOD Rec.*, 34(4):34–41.

Haas, L., Schwarz, P., Kodali, P., Kotlar, E., Rice, J., and Swope, W. (2001). Discoverylink: A system for integrated access to life sciences data sources. *IBM Systems Journal*, 40(2):489–511.

Hanson, E. N., Carnes, C., Huang, L., Konyala, M., Noronha, L., Parthasarathy, S., Park, J., and Vernon, A. (1999). Scalable trigger processing. In *In Proceedings of the 15th International Conference on Data Engineering (ICDE)*, pages 266–275.

Hellerstein, J. M. (1994). Practical predicate placement. In *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 325–335.

Hellerstein, J. M. (1998). Optimization techniques for queries with expensive methods. *ACM Transactions on Database Systems (TODS*.

Hellerstein, J. M. and Naughton, J. F. (1996). Query Execution Techniques for Caching Expensive Methods. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 423–434.

Hellerstein, J. M. and Stonebraker, M. (1993). Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 267–276.

Munagala, K., Srivastava, U., and Widom, J. (2007). Optimization of continuous queries with shared expensive filters. In *In PODS 07: Proc. of the twenty-sixth ACM SIGMODSIGACT-SIGART symposium on Principles of.* ACM.

Scheufele, W. and Moerkotte, G. (1998). Efficient dynamic programming algorithms for ordering expensive joins and selections. In *In Proc. of EDBT*, pages 201–215.

Zhang, Y., Yu, L., Zhang, X., Wang, S., and Li, H. (2012). Optimizing queries with expensive video predicates in cloud environment. *Concurr. Comput. : Pract. Exper.*, 24(17):2102–2119.