Generalized Scale Independence Through Incremental Precomputation

Michael Armbrust[†‡]
marmbrus@google.com

Eric Liang[‡]
ericliang@berkeley.edu

Tim Kraska[*‡]
tim_kraska@brown.edu

Armando Fox[‡]
fox@cs.berkeley.edu

Michael J. Franklin[‡]
franklin@cs.berkeley.edu

David A. Patterson[‡]
pattrsn@cs.berkeley.edu

[*]Brown University [†]Google, Inc. [‡]University of California, Berkeley

ABSTRACT

Developers of rapidly growing applications must be able to anticipate potential scalability problems before they cause performance issues in production environments. A new type of data independence, called *scale independence*, seeks to address this challenge by guaranteeing a bounded amount of work is required to execute all queries in an application, independent of the size of the underlying data. While optimization strategies have been developed to provide these guarantees for the class of queries that are scale-independent when executed using simple indexes, there are important queries for which such techniques are insufficient.

Executing these queries scale-independently requires precomputing results using incrementally-maintained materialized views. However, since this precomputation effectively shifts some of the query processing burden from execution time to insertion time, a scale-independent system must be careful to ensure that storage and maintenance costs do not threaten scalability. In this paper, we describe a scale-independent view selection and maintenance system, which uses novel static analysis techniques that ensure that created views do not themselves become scaling bottlenecks. Finally, we present an empirical analysis that includes all the queries from the TPC-W benchmark and validates our implementation's ability to maintain nearly constant high-quantile query and update latency even as an application scales to hundreds of machines.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

Keywords

Scalability; Scale Independence; Materialized view selection

1. INTRODUCTION

The ability to anticipate scalability problems is critical to fast-growing Internet services, such as Twitter and Facebook, both of whom have experienced long periods of exponential growth [15, 28]. When attempting to cope with this rapid influx of data, many scale-oriented developers have found that they prefer the straightforward pain of maintaining imperative implementations over the difficulty of identifying potentially expensive operations lurking beneath simple declarative expressions. Following this trend, NoSQL storage systems typically eschew declarative languages, sacrificing the productivity benefits these languages provide.

In our prior work, PIQL [4], we attempted to ensure that applications will perform predictably as they grow in popularity, while simultaneously preserving the many productivity benefits of the relational model. This syncretism was accomplished through the introduction of *scale independence* [5], a new type of data independence. As data sizes grow, even by orders of magnitude, a scale-independent system enforces strict invariants on the cost of query execution. Towards this end, PIQL incorporated an optimization algorithm that bounds the amount of work required to execute a given query. This technique was capable of handling SQL queries that could be answered scale-independently using only secondary indexes.

However, there are SQL queries, common in real world applications, where those previous techniques fall short. For these more complicated queries, on-demand execution could require reading unbounded amounts of data. For example, the popular online service Twitter needs to calculate the number of people following popular users. Executing this query on-demand could result in response time that grows with the size of the database. Fortunately, it is often possible to safely answer such queries at scale by leveraging *incremental precomputation*, effectively shifting some query processing work from execution time to insertion time.

We formally define two new classes of SQL queries where precomputation fundamentally changes the worst case execution cost at scale. Formalizing the characteristics of these classes allows us to construct a scale-independent view selection and maintenance system. PIQL's view selection system is unlike prior work on materialized views [2, 14, 17, 21], which attempted to minimize cost for a given workload. While these prior techniques select views that may speed up query execution on average, the overall performance of the application can unfortunately remain dependent on the size of the underlying database. Instead, PIQL focuses on ensuring scalability independent of data size and workload.

PIQL's optimizer creates incrementally-maintained materialized views (IMVs) only when it is possible to verify the cost of storage and maintenance for these views will not itself introduce a scaling bottleneck. Potential scaling issues are avoided using novel static analysis algorithms that ensure a given view obeys invariants on both the size of the view and the work required by incremental maintenance. We also describe schema analysis and query rewriting techniques that allow PIQL to detect hotspots and mitigate the resulting performance degradation. In summary, we present the following contributions:

- We formalize the invariants and implicit assumptions from prior work on scale independence.
- Using these invariants, we define four levels of scale-independent query execution, two of which were not covered by existing execution techniques.
- We present a scale-independent view construction algorithm along with static analysis techniques to bound the cost of storage and maintenance.
- We describe a mechanism for automatically detecting and mitigating common temporal hotspots using a combination of load balancing and parallel execution.
- We present an empirical study, including all of the queries from the TPC-W benchmark, of the high quantile latency for query execution and maintenance operations as our system scales to hundreds of nodes.

2. SCALE-INDEPENDENT QUERY PROCESSING

As first described by our prior work [5], a scale-independent query processing system seeks to aid developers of rapidly growing applications by guaranteeing predictable query response time, independent of the size of the database. One technique for ensuring this predictability is to maintain invariants on the operations performed and resources required for all queries in an application.

In this section, we first describe the four levels of scale-independent query execution. We then briefly review the optimization techniques used by the initial version of PIQL [4] for queries that can be executed on-demand, formalizing the invariant used to ensure that a given query performs a bounded number of operations in the worst case. We also formalize implicit assumptions made by PIQL's authors regarding the existence of a balanced partitioning for a given workload over a cluster of machines.

Building upon this foundation, we expand the discussion to include queries where previous techniques fall short, but where precomputation through the creation of an IMV can enable scale-independent execution. Since PIQL must ensure that these automatically created IMVs do not themselves threaten scalability, we place restrictions on the resources required for their storage and maintenance. Finally, we give an overview of the workflow used by PIQL to analyze all queries in an application and determine which scale-independent execution level should be used for each. This process results in a list of indexes and IMVs required.

2.1 Scale Independent Execution Levels

In order to help developers reason about the resource requirements of their application, we now define four levels of scale-independent query execution. Table 1 lists these levels along with the invariants (Sections 2.2 and 2.3) on execution, update and storage costs that must be enforced for each.

The first level, Scale Independance Level 0 (SI-0), includes trivially scale-independent queries (e.g. `SELECT 1`) as well as queries that can be answered in a scale-independent manner using the clustered index on the primary key (e.g. a query that does a lookup by primary key). For queries in SI-1, it is similarly possible to bound the amount of work performed while executing the query, but only through the creation of a secondary index. Section 2.2 reviews existing optimization techniques that statically analyze queries to determine if they can be executed using one of these two levels.

In contrast to the queries in SI-0 and SI-1, some queries could require an unbounded amount of work to execute on-demand, even after the creation of secondary indexes. Often, precomputation by creating an IMV can fix this problem, enabling scale-independent execution by shifting work to insertion time. However, unlike with simple indexes, a scale-independent view selection system must also consider the scalability of storage and maintenance costs, and we reason about these costs using additional invariants. Section 2.3 introduces these new invariants, which ensure that IMVs themselves do not become a scaling bottleneck. Queries fall into SI-2 if they can be executed in a scale-independent manner using an IMV that satisfies both of the new invariants.

	SI-0	SI-1	SI-2	SI-3
Execution (Inv 1)	I	D	D	D
Execution w/ indexes (Inv 1)	-	I	D	D
IMV Update (Inv 2)	-	-	I	D
IMV Storage (Inv 3)	-	-	I	I
IMV Parallel Updates (Relaxed Invariant 2)	-	-	-	I

Table 1: The four levels of scale-independent execution. 'I' and 'D' denote respectively that the cost of executing a query for a given resource is independent or dependent on scale of the application. A '-' implies the cost is not applicable, and thus the query is trivially scale independent in this dimension.

For each of the aforementioned execution levels it is assumed that the workload can be evenly balanced over all of the machines in the system. Section 2.5 explains in detail why this balanced partitioning is required to avoid the increased query response time associated with hotspots. Prior definitions of scale independence [4] failed to deal with the fact that the naive use of secondary indexes can violate this assumption in cases where there is temporal locality of insertions relative to the value being indexed (e.g. an index over the attribute `created_on`). Fortunately, these hotspots can often be mitigated by spreading new insertions across the cluster and periodically computing aggregate results in parallel. However, since this execution pattern requires relaxing the invariant on the total work performed by an update, we place queries that utilize this strategy in the final query level, SI-3.

2.2 Scale-Independent Optimization

To review our prior work: PIQL takes as input the set of parameterized queries Q used by an application. The optimizer analyzes all queries in Q to ensure that the database can grow while maintaining consistent performance.

For the purposes of scale-independence, database growth occurs over the following three dimensions:

$\|R\|$	the size of all base relations
Δ_{rate}	the update rate for all base relations
q_{rate}	the rate of read queries in the system

PIQL achieved scale independence across these dimensions by designing the optimizer to select only physical plans that perform a bounded number of I/O operations, independent of $|R|$ (i.e., the size of all base relations). In contrast to standard average cost minimization, PIQL's optimization technique prevents the selection of query plans that may perform well for most users but that could violate an application's Service Level Objective (SLO) for statistical outliers. It also allows the database to warn the developer of queries that pose a potential scalability problem and provide suggestions for resolving the issue before the query can cause SLO violations in production. The invariant that is maintained by this optimization technique can be formalized as:

Invariant 1. Let $Exec(q_i)$ denote the number of operations[1] performed in the worst case by a query and $c_{ops}^{q_i}$ be a constant for a given query q_i. A scale-independent optimizer will only create physical query plans such that:

$$\forall q_i \in Q \exists c_{ops}^{q_i} : Exec(q_i) < c_{ops}^{q_i}$$

A system can verify that an application will satisfy this invariant by performing a static analysis of the application's schema and queries. For example, uniqueness constraints ensure that a query that performs a lookup by primary key will return at most one result, and thus require a bounded number of operations to execute.

To expand the space of queries that can be verified as scale independent, it is possible to employ language extensions to standard SQL. For example, Data Definition Language (DDL) cardinality constraints (introduced in [4]) allow developers to specify restrictions on the relationships present in their application, which are enforced at insertion time.

To understand this optimization technique more concretely, consider a simple application that stores documents along with associated tags with the following schema:

```
Tags(docId, tag, timestamp) WITH CARDINALITY(tag, K)
Documents(docId, timestamp, text, ...)
```

Italicized columns form the primary key of a relation and the `WITH CARDINALITY` clause denotes a cardinality constraint K on the number of unique values of `tag` that can exist for any given `docId`. An example of a scale-independent query on this schema is the following parametrized SQL, which returns the set of tags for a given document.

```
SELECT * FROM Tags WHERE docId = <doc>
```

For any value of the `<doc>` parameter, the query can be executed by scanning a bounded range of the clustered primary key index on `Tags` and will return at most K tuples. Thus, the optimizer can guarantee that this query will never violate Invariant 1.

2.3 Scale-Independent View Selection

Not all queries can be answered scale-independently using only indexes, and in this section we describe how the addition of IMV selection to PIQL enables the scale-independent execution of many of these previously unsafe queries. For example, the following shows the `twoTags` query, which returns the five newest documents assigned two user-specified tags.

```
SELECT t1.docId
FROM tags t1, tags t2, documents d
WHERE t1.docId = t2.docId AND
      t1.docId = d.docId AND
      t1.tag = <tag1> AND t2.tag = <tag2>
ORDER BY d.timestamp LIMIT 5
```

While a secondary index on `Tags.tag` would allow efficient lookup of the documents for a given tag, such an index is not sufficient to enable the scale-independent execution of the `twoTags` query. The performance of the query is potentially dependent on the size of the data due to the fact that during any given execution an unbounded number of rows matching `tag1` might need to be scanned before finding five documents that also match `tag2` or vice versa. In practice, developers faced with a query such as this one often utilize a technique known as intersection precomputation or caching [19], where all two tag combinations for a document are computed ahead of time (analogous to the creation of a join index [27]).

While there has been significant prior work on leveraging precomputation through automatic materialized view selection [2, 17, 21] and incremental maintenance [1, 6, 7, 13, 25], these approaches have focused on minimizing average cost for a given workload, rather than ensuring consistent resource requirements as the database grows. As such, these techniques could create a view that may speed up query execution on average, while the absolute performance of the query remains dependent on the size of the underlying data. Simply executing faster is not sufficient to guarantee SLO compliance as an application explodes in popularity.

In contrast, PIQL must only create an IMV when it will allow a query to be answered scale-independently. In addition to determining the scalability of the query when it is run over the materialized view, we must also ensure that the resources required to incrementally maintain and store the created materialized views do not themselves threaten the scalability of the application. To better explain the potential scalability threat posed by the inclusion of materialized views we now formalize the additional invariants that must be maintained by a scale-independent view selection system.

Bounding Update Cost

A scale-independent view selection system must avoid IMVs whose update cost increases with the scale of the application. To this end, the optimizer checks that there is an upper bound on the number of operations required to incrementally update all indexes and views given an update to a single tuple in a base relation. Said formally:

Invariant 2. Let $Update(r_i)$ denote the number of operations performed in the worst case by index and view maintenance when updating a single tuple in r_i, and let $c_{ops}^{r_i}$ be a constant for r_i. A scale-independent optimizer will create views such that the total maintenance costs will never violate the following condition:

$$\forall r_i \in R \exists c_{ops}^{r_i} : Update(r_i) < c_{ops}^{r_i}$$

For queries in SI-3, we relax Invariant 2 to permit queries where serial work performed by a single machine is bounded, instead of the total work for a single update.

[1]Operations are restricted to those with nearly constant latency. For example, a bounded lookup given an index.

Bounding Storage

It is possible for the size of an IMV to grow super-linearly with the size of the base relations, for example, due to an unconstrained join. Therefore, PIQL ensures that the size of each view is at most a constant factor larger than one of the base relations present in the view. Said formally:

Invariant 3. Let V be the set of all created views required to answer the queries in Q and let $c^{v_i}_{storage}$ be a constant for a view v_i. Let r_{v_i} be a relation in R and $|r_{v_i}|$ denote the number of tuples in r_{v_i}. A scale independent system creates IMVs with linear storage requirements by ensuring that:

$$\forall v_i \in V \exists c^{v_i}_{storage}, r_{v_i} \in R : |v_i| < c^{v_i}_{storage}|r_{v_i}|$$

2.4 Query Compilation

Figure 1 shows the five phases of PIQL, which are used to determine how to scale-independently execute a query. In addition to validating the scalability of all queries in an application and creating necessary indexes and IMVs, PIQL will also tell developers which level each query falls into so they can reason about its resource requirements and update latency characteristics.

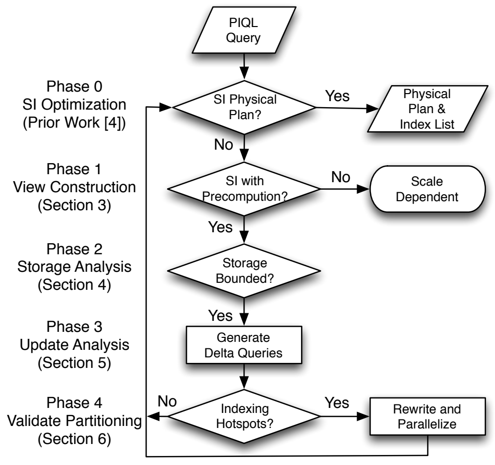
PIQL Query
Phase 0 SI Optimization (Prior Work [4])
SI Physical Plan?
Yes
Physical Plan & Index List
No
Phase 1 View Construction (Section 3)
SI with Precompution?
No
Scale Dependent
Yes
Phase 2 Storage Analysis (Section 4)
Storage Bounded?
Yes
Phase 3 Update Analysis (Section 5)
Generate Delta Queries
Phase 4 Validate Partitioning (Section 6)
No
Indexing Hotspots?
Yes
Rewrite and Parallelize

Figure 1: The phases of the scale-independent optimizer and view selection system.

Queries in level SI-0 or SI-1 are handled using prior techniques by generating a scale-independent physical execution plan (Phase 0), automatically creating indexes as needed. For cases where scale-independent physical plan for a query cannot be found by the optimizer, this paper describes how to expand a scale-independent relational system to answer the query instead by leveraging scale-independent precomputation (Phases 1-3). This view selection process occurs automatically through the creation of an IMV for queries that fall into SI-2. Additionally, PIQL ensures that common sources of hotspots are avoided (Phase 4), rewriting the query as a distributed staging step followed by a periodic parallel view refresh step. Since the resources required by the parallel view refresh step do not satisfy the strict form of Invariant 2, queries that require this transformation fall into SI-3.

2.5 Achieving Predictable Response Time

The bounds on execution cost enforced by Invariant 1 are useful not only for constraining resource requirements, but also for reasoning about performance. In [4], it was demonstrated empirically that bounding the number of operations performed by all queries in the worst case allows an application to achieve predictable response time as long as a scalable underlying storage system is used.

For example, many key/value stores can execute low-level operations such as `get(key)`, `getRange(prefix, limit)`, `put(key, value)` with consistent performance, even at high quantiles, by using a combination of autonomic techniques and over-provisioning [11, 26]. Since each query or update performs only a bounded number of these operations in the worst case, it is possible to reason about the probability distribution of the worst case execution time of each query [4]. This worst case reasoning is especially valuable for developers of interactive applications, who care about the response time for every user of their system.

While [4] demonstrated that it is possible to achieve nearly constant latency while scaling up, it is important to understand the implicit assumption that enables this predictability. Specifically, achieving consistent performance as the amount of data and number of machines grow is only possible when the growing workload can be spread evenly across machines in an ever growing cluster.

If no balanced partitioning of the workload exists, then eventually a single partition will become overloaded by the increased workload. As queues build on the overloaded machine, the latency for requests to this server will grow and eventually affect the overall response time of the application. For example, a hotspot will eventually occur when there is a query that requires a secondary index over the tuple creation timestamp field of a table. Maintaining this index naively would eventually result in a hotspot at the server holding the partition corresponding to the current time.

Section 6 describes how PIQL automatically avoids the creation of such indexes, instead utilizing workload balancing coupled with a periodic parallel collection step.

3. VIEW SELECTION

The first phase of PIQL's view selection system, view construction, is invoked when the optimizer is unable to find a scale independent physical plan for a query. Since this query does not fall into either SI-0 or SI-1, the optimizer will instead try to answer the query by creating an IMV. Note that this is only the first step in scale-independent view selection and merely constructs a view that *could* be used to answer the query while satisfying Invariant 1. The algorithms in this section do not yet ensure that this view will meet the other invariants regarding storage and maintenance costs. The general form of the IMVs created is as follows:

```
CREATE VIEW <viewName>
SELECT $A_{view}$ [$A_{agg}$]
FROM $r_1, r_2, ...r_n$
WHERE $P_{view}$ [GROUP BY $A_{eq}$]
```

A_{view} is an ordered list of the attributes projected by the IMV, while A_{agg} contains any aggregate expressions. The relations r_1 to r_n are the base relations present in the original query, and P_{view} is the set of predicates for the IMV. In the case of queries with aggregate expressions, a `GROUP BY` clause is also added to the IMV definition.

The construction algorithm ensures the created IMV will allow scale-independent execution of the original query by assuming that the underlying storage structure allows scale-independent access to tuples in the view given a prefix of the attributes present in A_{view}. Examples of acceptable storage systems include B-Trees and range-partitioned distributed key/value stores. We adopt the convention that the SQL expression used to define an IMV must specify not only which tuples are present in the view but also the ordering of attributes in the clustered index used to store the view. Thus, the definitions of created IMVs effectively define the spatial locality of the precomputed tuples.

3.1 View Construction Without Aggregates

The view construction algorithm presented in this section handles *select-project-join* queries with *conjunctive* predicates. Section 3.2 expands this algorithm to enable support for queries with aggregate expressions. At a high level, this algorithm is solving a variant of the view selection problem where the result of the target query must be computable by scanning a contiguous section of the view for any value of the runtime parameters. We start by describing how predicates that include runtime parameters are handled. Next, we describe the checks employed to ensure that the result of the original query can be obtained by scanning a contiguous section of the selected view. Finally, we describe the construction of the final view definition.

View construction starts by parsing predicates of the form *attr* `= <parameter>` or *attr* $\{<, >, \leq, \geq\}$ `<parameter>`. Using these predicates, the algorithm constructs the sets A_{eq} and A_{ineq}, which contain attributes occurring in equality and inequality predicates respectively. These attribute sets are added to the prefix of the projection of the view, allowing the predicates of the original query to be evaluated efficiently using an index scan.

As a concrete example, for the `twoTags` query (Section 2) the set A_{eq} contains `{t1.tag, t2.tag}`, and A_{ineq} is empty.

Next, Algorithm 1 checks to ensure that the answer to the original query for any set of runtime parameters will be a contiguous set of tuples in the view. The requirements for this condition are as follows: Any number of attributes may be filtered by equality with a runtime parameter. In contrast, at most one attribute can be filtered by an inequality with a parameter, though this attribute may appear in more than one predicate (i.e., in the case of interval queries). This limitation is due to the fact that computing the intersection of two inequality predicates against different attributes may involve scanning over an arbitrary number of tuples in the view, and thus could violate Invariant 1. Similarly, any number of attributes may be specified in the `ORDER BY` clause, but this ordering must be prefixed by the inequality attribute, if one exists.

Algorithm 1 Confirming Adjacency of Result Tuples

```
1: if $|A_{ineq}| > 1 \vee (|A_{order}| > 0 \wedge A_{order}[0] \neq A_{ineq})$ then
2:     return $false$
3: end if
4: return $true$
```

If Algorithm 1 returns successfully, it then creates the set P_{view}. This step is accomplished by removing any predicates that involve a runtime parameter and then simplifying any redundant equality predicates.

To avoid changing the meaning of the query, it is important that this procedure does not inadvertently discard any transitive equality constraints present due to parameters that appear multiple times in the query. For example, given a query with the predicates a_1 `= <p1>` and a_2 `= <p1>`, P_{view} must contain the predicate $a_1 = a_2$. We account for these transitive equalities by generating predicates for the view from the equivalence classes defined by the equality predicates in the original query. Inequality predicates, in contrast, are copied directly from the original query. These predicates will eventually be turned into inequalities with parameters during delta query calculation (Section 5). Therefore, the rules regarding multiple attributes participating in inequalities still apply, and thus the creation of the view could be later be rejected due to a lack of a scale-independent maintenance strategy.

Algorithm 2 Generating View Predicates

```
1: $P$ := set of all conjunctive predicates in the query of the
   form $v1$ $op$ $v2$
2: EquivalenceClasses(P) := set of equivalence classes under P
3: $P_{view} \leftarrow \{\}$
4: for all $X \in EquivalenceClasses(P)$ do
5:     $prevAttr := \varnothing$
6:     for all $v \in X$ do
7:         if $!isParam(v)$ then
8:             if $prevAttr \neq \varnothing$ then
9:                 $P_{view} := P_{view} + Equality(prevAttr, v)$
10:            end if
11:            $prevAttr := v$
12:        end if
13:    end for
14: end for
15: for all $p \in P$ do
16:     if $isInequality(p.op)$ then
17:         if $!isParam(p.v1) \wedge !isParam(p.v2)$ then
18:             $P_{view} := P_{view} + p$
19:         end if
20:     end if
21: end for
```

Algorithm 2 describes this process of creating P_{view}. Taking as input the set of conjunctive predicates (Line 1), the algorithm starts by partitioning values found in P into equivalence classes (Line 2). Next, for every equivalence class (Line 3), the algorithm adds an equality predicate to P_{view} for each attribute pair in the class (Line 4-11). Finally, the algorithm copies inequality predicates involving non-parameters into P_{view} (Line 15-21).

As an example, consider again the `twoTags` query. The predicates in this query define three equivalence classes:

```
{t1.docId, t2.docId, d.docId}, {t1.tag, <tag1>},
{t2.tag, <tag2>}
```

From these classes the following is produced for P_{view}:

```
{t1.docId = t2.docId, t1.docId = d.docId}
```

Once P_{view} has been constructed, the system next creates A_{view}. A_{view} contains the attributes in A_{eq} and A_{ineq} as well as any remaining key attributes from the top-level tables present in the original query. These key attributes are added to the projection in order to ensure the view can be efficiently maintained using production rules, as proposed by Ceri and Widom [7]. Specifically, the view selector ensures that there will be no duplicate tuples in the view and that all top-level table references are safe. Additionally, to avoid unnecessary redundancy, any key attributes that are unified by an equality predicate in the original query will only appear once in the projection of the view.

Algorithm 3 Choosing View Keys

```
1: $R :=$ the set of relations present in the original query
2: $A_{keys} := \{a : r \in R, a \in keyAttrs(r)\}$
3: $A_{view} \leftarrow []$
4: $A_{covered} \leftarrow \{\}$
5: function EquivalenceClass($P, a$)
6:     return all values in P unified with $a$ due to equality
       predicates, including $a$ itself
7: end function
8: for all $a \in A_{eq} + A_{ineq} + A_{order} + A_{keys}$ do
9:     if $a \notin A_{covered}$ then
10:        $A_{view} := A_{view} + a$
11:        $A_{covered} := A_{covered} \cup EquivalenceClass(P, a)$
12:    end if
13: end for
```

Algorithm 3 describes the process used to populate A_{view} and starts by initializing A_{keys} to be the set of key attributes for all top-level tables from the original query (Line 2). Next, it initialises A_{view}, the ordered list of attributes that will appear in the view, and $A_{covered}$, the set of attributes already represented in the view definition considering unification, to be empty (Lines 3-4). Then, it iterates over the ordered concatenation of the attribute sets (Line 8) and adds to the view any attributes not yet present in the cover set $A_{covered}$ (Line 10). To prevent redundant values from appearing in the view, when an attribute is added, its entire equivalence class is added to $A_{covered}$ (Line 11).

Applying Algorithm 3 to the `twoTags` query, the attributes `t1.tag`, `t2.tag`, `d.timestamp`, and `d.docId` are selected. The first two attributes come from A_{eq}, `d.timestamp` comes from A_{order}, and `d.docId` from A_{keys}. By placing attributes from A_{eq} first, the view selector ensures that tuples satisfying the original query can be located by a prefix of the keys in the view. Including `d.timestamp` next ensures that the relevant tuples in the view will be sorted as specified by the `ORDER BY` clause. Finally, `d.docId` allows for safe view maintenance and the retrieval of the actual document.

Continuing the `twoTags` example, the following materialized view is created by the view constructor:

```
CREATE VIEW twoTagsView
SELECT t1.tag as t1tag, t2.tag as t2tag,
       d.timestamp, d.docId
FROM Tags t1, Tags t2, Documents d
WHERE t1.docId = t2.docId AND
      t1.docId = d.docId
```

Once the view has been constructed for a given query, the query is rewritten by replacing the top-level tables with the view and renaming any attributes to their equivalent attribute in the view. If there are any attributes that are present in the original query, but not in the view, they can be retrieved either by joining the view with the base relation on the keys that are present, or by adding the missing attributes to the view definition. The former will require more computation at query time while the latter will require more storage for the view. Since the method does not affect the scale independence of the query, the system decides which technique to use based on the predicted SLO compliance of the resulting query.

This final step rewrites the `twoTags` query to use the materialized view as follows:

```
SELECT t1.docId
FROM twoTagView
WHERE t1tag = <tag1> AND t2tag = <tag2>
ORDER BY timestamp LIMIT 5
```

3.2 View Construction With Aggregates

PIQL's view selection system is also capable of handling many queries that contain aggregates in the target list. In this subsection, we describe both the class of aggregates that are supported and the alternative view selection algorithm used when an aggregate is present in a query.

3.2.1 Scale-Independent Aggregates

The class of scale-independent aggregates is defined in part by the storage requirements for partial aggregate values. Using the categories of aggregates first defined by Gray et al. [12], PIQL can safely store partial aggregate values for both *distributive* aggregates (such as `COUNT` or `SUM`) and *algebraic* aggregates (such as `AVERAGE` and `VARIANCE`). Both of these categories have partial state records of fixed size. In contrast, *holistic* aggregates like `MEDIAN` can require an unbounded amount of partial state to be stored and thus conflict with our goal of scale independence.

Bounding the storage required for each partial aggregate value alone is not sufficient, as we must also ensure that efficient incremental maintenance of aggregated values is possible. Towards this end, we also require updates to the aggregate be both associative and commutative. While both `MIN` and `MAX` are distributive aggregates, updates to them do not always commute in the presence of deletions. For example, when the maximum value from a given group is deleted, the only way to update the aggregate is to scan over an unbounded set of tuples looking for the new maximum value.

3.2.2 View Selection with Aggregates

When PIQL's view selection system detects an aggregate in the projection of a scale-dependent query, it uses a slightly modified view construction algorithm. We explain these modifications in four parts.

Ordering Views containing aggregates cannot have any inequalities with parameters, as these could require unbounded computation to maintain. Additionally, since each query containing an aggregate will return only one tuple, there is no `ORDER BY` clause. These two changes eliminate the need for Algorithm 1.

Keys Views containing aggregates are maintained using techniques analogous to a counting solution [13]. Thus, it is not necessary to use Algorithm 3 to add keys for safe maintenance.

Aggregate Expressions The view selection algorithm must add partial aggregate values A_{agg} to the created view. In the case of distributive aggregates, only the aggregate expression itself must be added, while for algebraic aggregates more information may be required to ensure efficient incremental maintainability. For example, `AVERAGE` is computed by keeping both a `SUM` and a `COUNT`.

Group By Clause All of the attributes in A_{eq} are added to the group by clause of the created view.

The following is an example of how this modified algorithm would select a view for the `countTags` query, which calculates the number of documents assigned a given tag.

```
SELECT COUNT(*)        CREATE VIEW docsPerTag
FROM Tags              SELECT tag, COUNT(*) as cnt
WHERE tag = <tag>      FROM Tags t
                       GROUP BY tag
```

3.3 Views for Window Queries

PIQL can handle many queries that operate over windows of data, and the techniques for handling these queries fall into two categories. For queries that operate over a fixed size tuple window, an index can be created on insertion timestamp. Since the tuple window bounds the number of tuples that will need to be retrieved from this index, scale independent optimization can then be performed on the rest of the query using standard techniques. Section 6 discusses the special consideration required when creating such indexes to avoid hotspots as the size of the database scales.

Aggregate queries that operate over a time window are handled by prepending an *epoch* identifier to the beginning of the view. The epoch identifier calculated using the following formula: $timestamp - (timestamp \mod windowSize)$. For example, consider modifying the `countTags` query from the previous section to count tags for a sliding window of length one minute. PIQL would create the following view (assuming timestamp is measured in milliseconds).

```
CREATE VIEW docsPerTagWindowed
SELECT (timestamp - (timestamp % (60*1000)),
       tag, COUNT(*) as count
FROM Tags t
GROUP BY (timestamp - (timestamp % (60*1000)), tag
```

Queries where the window size is larger than the slide amount can also be handled but will result in updates to all relevant epochs. Stale epochs can be garbage collected.

4. BOUNDING STORAGE COSTS

Once a candidate view has been produced by the selection algorithm described above, Phase 2 performs a static analysis of the maximum possible storage requirements. If the analysis determines that the view could grow super-linearly relative to the size of the base relations, the view is rejected, as its creation might violate Invariant 3.

PIQL's view size analysis utilizes dependency information from both the schema and the view definition. Note that the dependencies defined in this section subsume standard functional dependencies, where the latter can be represented as a cardinality dependency of weight one.

At a high level, the algorithm bounds the maximum size of a view by determining how many degrees of freedom remain after taking into account all of the tuples from a single relation in the view definition. In doing so, the algorithm determines whether the dependencies present are sufficient to ensure that the view is bounded in size by a constant factor relative to at least one base relation.

4.1 Enumerating Dependencies

The analysis starts by constructing the list of all dependencies for a given view definition. These dependencies are generated by analyzing the view as well as the schema of the application using the following four rules:

1. $(keyAttributes) \rightarrow (otherAttributes)$
   Add a dependency of weight one to represent the functional dependency due to the primary key's uniqueness constraint.
2. $(keyAttributes) \xrightarrow{cardinality} (constrainedFields)$
   Add a dependency for each relation that has a cardinality constraint declared in the schema, weighted by the cardinality of the constraint.
3. $attribute_1 \leftrightarrow attribute_2$
   Add a bidirectional dependency of weight one for each attribute pair present in an equality predicate in the `WHERE` clause of the view definition.
4. Fixed Value $\rightarrow attr$
   Add a directional edge from a special fixed node to any attribute that is fixed by an equality predicate with a literal (e.g. $attr$ = true). This dependency is always implied, independent of what other dependencies are being considered.

Take, for example, the view definition of the `twoTags` query from the previous section. Given this view definition, the algorithm will produce the following list of dependencies. We omit trivial dependencies for the sake of brevity.

$$d.docId \rightarrow (d.timestamp, \ldots) \quad (1)$$

$$t1.docId \xrightarrow{K} t1.tag \quad (2)$$

$$t2.docId \xrightarrow{K} t2.tag \quad (3)$$

$$t1.docId \leftrightarrow t2.docId \quad (4)$$

$$t1.docId \leftrightarrow d.docId \quad (5)$$

First, dependency 1 is added to the set due to the primary key of the document relations. Next, dependencies 2 and 3 are added to the set due to the cardinality constraint that each document may have no more than K tags. Finally, dependencies 4 and 5 are added as a result of the equality predicates present in the view definition.

4.2 Bounding Size Relative to a Relation

The following algorithm determines if the maximum size of the view is linearly proportional to one of the relations present in the query, using the dependency list generated by the above rules. This analysis is performed by finding a relation present in the query such that all attributes present in the view are functionally dependent on the primary key of the selected relation. If independent attributes remain after the inclusion of all possible dependencies, then a bound on the size of the IMV does not exist relative to that relation. If no relation can be found such that all attributes are functionally dependent on its primary key, then the view is rejected due to a possible violation of Invariant 3.

Algorithm 4 describes this process formally, and takes as input the set of all relations present in the view and all dependencies for the IMV (Lines 1-2). It returns a boolean value indicating if there is an upper bound on the size of the view due to these dependencies. The algorithm iterates over all of the relations present in the view definition. For each relation, the set of attributes functionally dependent on this relation a_{dep} is initialized to the key attributes of the relation (Line 4). Then, the algorithm iteratively selects the set of attributes a that are functionally dependent on the attributes in a_{dep} given D but are not yet present in a_{dep} (Line 6). These new attributes are then added to a_{dep}

(Line 7). If a_{dep} now includes all attributes from the view, we know the view size is bounded relative to the relation r, thus, the algorithm returns $true$ (Lines 8-10). The iteration stops if at any point there are no more attributes to add to the set (Line 11). If no bound can be found for any relation, the algorithm returns $false$ (Line 13).

Algorithm 4 Bounding Maximum View Size

```
 1: $R :=$ the set of all relations present in the view
 2: $D :=$ the set of dependencies for the IMV
 3: for all $r \in R$ do
 4:     $A_{dep} := keyAttrs(r)$
 5:     repeat
 6:         $A := \{a \mid a \in attrs(R), a \notin a_{dep}, D \models A_{dep} \to a\}$
 7:         $A_{dep} := A_{dep} \cup A$
 8:         if $A_{dep} \supseteq attrs(R)$ then
 9:             return $true$
10:         end if
11:     until $|A| = 0$
12: end for
13: return $false$
```

To understand this process more concretely, consider again the definition of the IMV created by our system to answer the `twoTags` query. Algorithm 4 will start by selecting the Tags (t1) relation. This initializes a_{dep} to `{t1.id, t1.tag}`. On the first iteration, it will add `{t2.docId, d.docId}` to a_{dep} due to dependencies 4 and 5 respectively. Then, on the second iteration, it will add all remaining attributes due to dependencies 1 and 3. At this point, since all attributes in the query are in a_{dep}, the algorithm will return $true$.

Note that Algorithm 4 would return $false$ were it not for the cardinality constraint on the number of tags per document (functional dependencies 2 and 3). To understand how this schema modification could cause the `twoTagView` definition to violate Invariant 3, consider the degenerate case of a database with only a single document. As the number of tags increases, the size of the view would grow quadratically.

4.3 Views with GROUP BY

When the view contains a `GROUP BY` clause, we must modify the procedure for determining if the storage required by the view is bounded. This modification is a result of the fact that the `GROUP BY` effectively collapses many tuples down to those with unique values for the attributes in the `GROUP BY`. Thus, instead of requiring attributes from all relations to be covered by the dependencies, it is sufficient to have dependencies that cover only the attributes being grouped on. This change can be implemented simply by substituting $attrs(R)$ with the set of attributes present in the `GROUP BY` on Line 8 of Algorithm 4.

5. BOUNDING MAINTENANCE COST

Once the view selection system has produced a candidate view and verified that a bound exists for the storage required by the view, Phase 3 ensures the existence of an upper bound on the number of operations required by incremental maintenance given a single update to any of the relations present in the view. In this section, we first review the standard techniques used to perform incremental maintenance. We then explain the analysis performed to ensure that the total number of operations required by this mechanism will be bounded.

5.1 Maintenance Using Production Rules

PIQL performs incremental maintenance using production rules [7] that execute each time a base relation is modified. At a high level, the production rules update the view by running a *delta query*. This delta query calculates all of the tuples that should be added or removed from the view due to a single tuple insertion or deletion. An updated tuple is processed as a delete followed by an insert.

Since PIQL's view construction algorithm ensures the safety of incremental maintenance, the delta query for an update can be derived by substituting the updated relation with the single tuple being inserted or deleted. In order to understand the delta query's derivation more concretely, consider the IMV `twoTagsView` when a new tag is inserted. PIQL's view selection system would calculate the first delta query by substituting the relation `t1` with the modified tuple in the view definition. Notationally, we represent the values of inserted or deleted tuple as `<parameters>` to the delta query. This substitution produces the following rule which will be run anytime a tuple is added to the `Tags` relation:

```
CREATE RULE newTag ON INSERT Tags
INSERT INTO twoTagsView
SELECT <tag>, t2.tag as t2tag, d.timestamp, d.docId
FROM Tags t2, Documents d
WHERE t2.docId = <docId> AND d.docId = <docId>
```

Update rules must be created for all of the relations present in the view definition, including another rule for the second instance of the `Tags` relation. Due to the presence of a self-join, the inserted tuple must also be joined with itself. We omit the other delta queries from the paper for brevity.

5.2 Maintenance Cost Analysis

Given all of the delta queries for a view, we must verify that none of them threaten the scalability of the application by requiring an unbounded number of operations during execution. Fortunately, since these delta queries are represented as SQL queries, we can reuse the optimization techniques from prior work [4] to perform this analysis. If the optimizer is able to find scale-independent plans for all of the delta queries, then we can certify that the addition of the view will not cause a violation of Invariant 2.

For example, consider the delta query `newTag` from the previous subsection. Due to the cardinality constraint on the number of tags per document, this query can always be executed by performing a single sequential scan.

Since verifying the scalability of the delta queries involves invoking the optimizer again, it is possible that the only scale-independent physical plan for a delta query will also require the creation of an index or materialized view. Thus, in some cases the creation of a materialized view could result in multiple *recursive* invocations of the view selection system. Fortunately, we can be assured that this recursion will always terminate since the degree of the query will decrease with each successive derivation [16].

5.3 Updating Aggregates

Materialized views that contain aggregates are also maintained using delta queries with one important distinction. Delta queries for aggregate functions return a list of updates to possibly existing rows instead of tuples that will be added or removed from the IMV. Notationally, we represent the update that will be applied to a given field in the view as an expression in the `SELECT` clause prefixed by a `+`.

The delta queries themselves are derived using rules similar to those used for other queries. Specifically, the inserted tuple is substituted for the relation being updated and simplified, leveraging the distributivity properties of joins and aggregates [16]. For example, considering the view for the `countTags` query, the following delta query is used for maintenance:

```
CREATE RULE ON INSERT INTO Tags
UPDATE twoTagsCount
SELECT <tag>, t2.tag, +1
FROM Tags t2
WHERE t2.docId = <docId>
```

When a new tag is inserted for a document, this query increments the count of all combinations of this tag and others already present for that document. The rules for deletions are symmetric and thus omitted for brevity.

6. AVOIDING COMMON HOTSPOTS

Preventing workload hotspots a critical when attempting to maintain consistent performance in a distributed query execution system. While clearly it is not possible to predict and avoid all possible hotspots, PIQL is able to detect some common cases using schema annotations provided by the developer. Based on these annotations, PIQL will avoid creating indexes that could result in a hotspot, and will instead suggest views that spread the insertions across the cluster. In this section we describe an addition to standard DDL and the technique used to rewrite potentially problematic queries.

6.1 DDL Annotations

PIQL allows developers to annotate columns whose values exhibit strong temporal locality with respect to insertions. Figure 2 shows the canonical example of such locality, an index over the creation timestamp for a given record. Since all records created within a short time period will have similar values for this attribute, all updates to an index ordered over this attribute will be routed to the same partition. While this concentration of updates will not result in performance issues at a small scale, it is in direct conflict of our goal of predictable performance as the system grows.

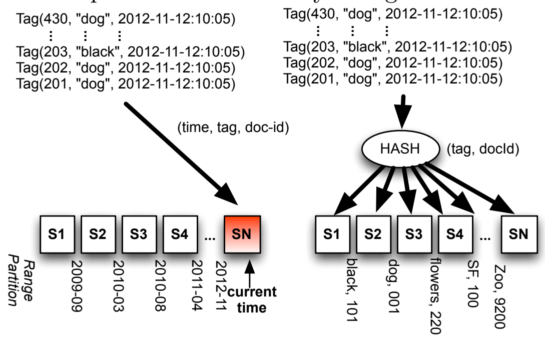
Tag(430, "dog", 2012-11-12:10:05)
Tag(203, "black", 2012-11-12:10:05)
Tag(202, "dog", 2012-11-12:10:05)
Tag(201, "dog", 2012-11-12:10:05)
(time, tag, doc-id)
S1
S2
S3
S4
...
SN
Range Partition
2009-09
2010-03
2010-08
2011-04
2012-11
current time
Tag(430, "dog", 2012-11-12:10:05)
Tag(203, "black", 2012-11-12:10:05)
Tag(202, "dog", 2012-11-12:10:05)
Tag(201, "dog", 2012-11-12:10:05)
HASH
(tag, docId)
S1
S2
S3
S4
...
SN
black, 101
dog, 001
flowers, 220
SF, 100
Zoo, 9200

Figure 2: Indexes over timestamps can result in hotspots, denoted by the shaded server. In contrast, PIQL chooses to distribute insertions over all machines in the cluster.

Developers can warn the optimizer to avoid the creation of these hotspot prone indexes by using the `TEMPORAL` keyword. For example, consider the following modification to the `Tags` schema, first introduced in Section 2.

```
Tags(docId, tag, timestamp TEMPORAL) WITH ...
```

6.2 Query Rewriting

When PIQL detects a hot-spot prone index, it instead creates materialized views that can be used to answer the query using a two step process. As an example of a case where this process would occur, consider the following query, popularTags, which returns the most popular tags out of the most recent 5000 insertions.

```
SELECT tag, COUNT(*)
FROM (SELECT * FROM Tags
      ORDER BY timestamp
        LIMIT 5000)
GROUP BY tag
```

Using only prior techniques, the optimizer would classify this as SI-1 and attempt to execute it on-demand with an index over `Tags.timestamp`. This execution plan would be problematic at scale, however, as maintaining this index in a range partitioned system would eventually lead to an overloaded partition and subsequently high query latency.

Instead of creating this hot-spot prone index, PIQL will instead create a view that is hash partitioned by the primary key of the relation. In the example below, this partitioning can be seen though the `HASH` keyword at the beginning of the view definition. Within each partition the records are sorted by the `TEMPORAL` attribute (i.e., `timestamp` in this example). Any remaining required attributes are appended to the end of `SELECT` clause. This procedure results in the following materialized view for the `popularTags` query.

```
CREATE VIEW popularTagStaging
SELECT HASH(tag, docId), timestamp, tag
FROM Tags
```

Next, the original query is rewritten as a periodically updated materialized view by substituting the original relation with the hash-partitioned materialized view. For the `popularTags` query this transformation results in the following SQL.

```
CREATE VIEW popularTags PERIODIC 1 MIN
SELECT tag, COUNT(*)
FROM (SELECT * FROM popularTagStaging
      ORDER BY timestamp
        LIMIT 5000)
GROUP BY tag
```

The optimizer will choose a physical plan that executes the subquery in parallel on each partition. Since the tuples in each of the partitions are sorted by the desired ordering attribute, each partition will only need to scan over 5000 tuples in the worst case. Therefore, the total amount of work that needs to be performed serially has a constant upper bound. However, since the total amount of computation is now proportional to the number of machines in the cluster, this query now falls into SI-3, unlike the twoTags query from the previous sections, which falls into SI-2. Section 8.2.2 demonstrates that even though the total amount of work grows with the size of the cluster, the increased parallelism allows us to execute the query with only minor increases in the periodic update latency.

7. CONSISTENCY

PIQL's techniques are compatible with a variety of consistency guarantees, including online maintenance [20, 23] or deferred view maintenance [24, 9]. For applications with strong consistency requirements, a locking-based view maintenance mechanism can be used [23, 20]. However, depending on the data access patterns such locking-based techniques might create contention points and therefore potentially violate our goal of predictable performance at scale. As an alternative to locking-based techniques, many deferred view maintenance techniques, such as 2VNL [24], perform optimistic view maintenance. Finally, eventually consistent view maintenance techniques [23] provide the lowest overhead per update, but often at the price of a few concurrent updates never being reflected in the view (see also view maintenance anomalies [18]).

The current implementation uses the simplest technique, relaxed view maintenance. We, however, ensure that all updates are eventually reflected in the view using the following procedure: Delta queries execute under relaxed consistency, while background batch jobs check for and repair any inconsistencies that may arise by periodically reconstructing the entire materialized view. The frequency of the execution of these batch jobs can be tuned to meet the needs of a specific application. This approach provides a balance between low-latency updates, high availability, and consistency.

Clearly, the efficiency of PIQL could be improved by only checking for inconsistencies in those sections of the view that have changed recently as done in [24]. We believe this optimization as well as other mechanisms for eventually-consistent, distributed incremental view maintenance represent interesting future research problems.

8. EXPERIMENTS

In order to understand the effect the scale-independent invariants have on the latency of both query execution and view maintenance, we ran two sets of experiments. The first, a micro-benchmark based on the `twoTags` example query (Section 2), demonstrates the need for an IMV when answering a query in SI-2, as well as showing bounded latency for both query execution and the incremental maintenance of the view. The second experiment demonstrates the scalability of our system under a more holistic workload by running all queries from the TPC-W benchmark, using the modifications and IMVs suggested by PIQL's optimizer.

Experiments were run using Amazon EC2 m1.large instances, which have 7.5 GB of memory and CPU power roughly equivalent to a two-core, 2GHz processor. SCADS [5] was used for the underlying storage system and provisioned for each experiment such that all nodes were responsible for the same amount of data and workload.

8.1 TwoTags Benchmark

Without an IMV, the `twoTags` query can often be expected to return quickly, but common data access patterns can result in arbitrarily slow response times. For example, consider a case where a large number of documents are assigned `tag1` but few documents are also assigned `tag2`. With only an index, the system will need to read many of the documents with `tag1` during query execution, resulting in query latency that grows proportionally with the number of tuples touched. To validate whether this problematic scenario arises in practice, we construct datasets at varying scales in which document tags are sampled from a Zipf distribution (n=2000, s=0.1). We chose this distribution to approximate the frequency of tags in a social context [10, 22].

We measure the performance of the `twoTags` query at scale by partitioning the `Tags` relation as well as its materialized view across the cluster by key range, with two replicas per partition. Each experiment begins with 200,000 tags bulk loaded to each partition. A cardinality constraint of 10 tags per document is enforced, with an average of four tags per document initially. For each data point, we combine the results from multiple runs across different EC2 cluster instantiations, and discard the first run of any setup to mitigate JIT warm-up effects.

A 2:1 ratio of storage to client nodes is maintained across cluster sizes, with each client utilizing two reader and two writer threads to issue requests to storage concurrently. Since writes are significantly more expensive than reads, this results in a read-write ratio of approximately 25 to 1.

8.1.1 On-Demand vs. Materialization

Figure 3 shows that with on-demand execution, the 99th percentile latency grows linearly with the size of the data, and the query read latency quickly rises to over a second. The significant increase in response time clearly demonstrates the danger of allowing queries with scale-dependent plans to run in production. However, when the same query is answered using the IMV, the response time remains nearly constant, leveling off at a 99th percentile latency of 8ms.

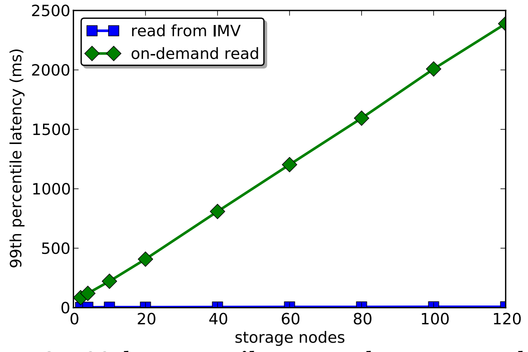

Figure 3: 99th-percentile query latency as cluster size grows with and without an IMV.

8.1.2 Cost of Incremental View Maintenance

Now that we have demonstrated bounded query latency by shifting some of the computational work to insertion time, we must now verify that we have not negatively impacted the scalability of writes to the application.

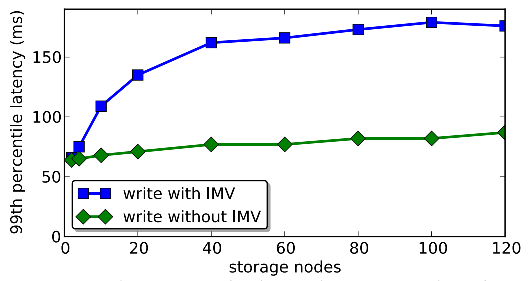

Figure 4: Write completion time remains bounded even with the overhead of incremental maintenance.

While the incremental view maintenance could be performed asynchronously, returning to the user immediately after the write is received, we modified the system to perform all maintenance synchronously for the purpose of this experiment. Figure 4 shows the effect that the maintenance of the materialized view has on write latency.

While latency of updates to the `Tags` relation is impacted by view maintenance, the effect is relatively small ($\sim$ 110ms in the worst case) and remains virtually constant for cluster sizes larger than 40 nodes. The initial increase in write latency as the cluster grows is due to the fact that the number of partitions in the system is small with respect to the cardinality constraint on tags. Specifically, for a smaller cluster, fewer partitions are often contacted per write, since some writes will go to the same machine. However, since Invariant 2 bounds the number of writes that will be performed in the worst case, eventually the maximum distribution is reached and the performance effect levels off.

8.2 TPC-W Benchmark

TPC-W[2] is a standard benchmark based on an online bookstore. Unlike prior work [4], where some TPC-W queries were excluded from scalability analysis, we include all web interactions in our evaluation. PIQL's automatic creation of IMVs allows for full execution of the benchmark, while still achieving consistent 99th percentile response time.

Two web interactions required slight modifications to avoid naive workload hotspots. The first, the BestSellerWI, returns the top 50 most popular items from the last 3333 orders. The second, AdminConfirmWI, returns the 5 most common items co-purchased with a specified item from the last 10,000 orders. Since executing either of these interactions on-demand would require an index over the creation time for a given order, PIQL suggested that we precompute the answer using periodically refreshed materialized views.

We went further and changed the tuple windows to instead calculate the result over hour-long time windows. TPC-W was initially designed to be run on a single machine and thus the queries were not written expecting arbitrarily high order rates. Since a time window both provides a more semantically consistent result as the system grows and requires strictly more work to maintain than simple tuple windows, we chose this implementation for our evaluation.

8.2.1 Query and Update Performance

After applying the scalability modifications, PIQL's optimizer was able to find query and update plans that satisfy all scale-independent invariants. Figure 5 shows the 99th percentile response time for the BuyConfirm web interaction. Since this interaction is the only one that actually places orders, all incremental view maintenance is performed during this action. Similar to the `twoTags` query, the strict upper bound on the number of operations performed in the worst case causes the increase in response time to eventually level off. Figure 6 shows that PIQL provides consistent 99th percentile response time for all web interactions.

Without view maintenance, the BuyConfirm interaction has a 99th percentile response time of 100ms. This means that, in the worst case, view maintenance adds nearly two seconds of latency to this web interaction. Fortunately, in a production system, this maintenance could be performed asynchronously and thus not affect end user experience.

A large majority of query processing time is due to network latency. For all TPC-W queries, we measured this

[2]http://www.tpc.org/tpcw/

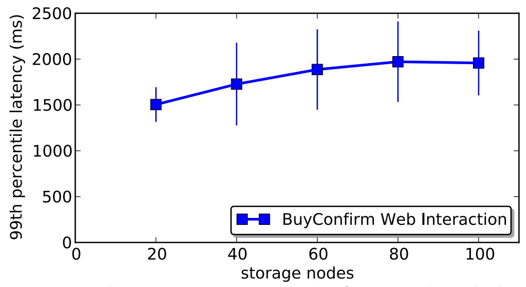

Figure 5: The response time for maintaining the IMVs for the TPC-W workload increases initially, but eventually levels off due to the limitations imposed by the scale-independent invariants.

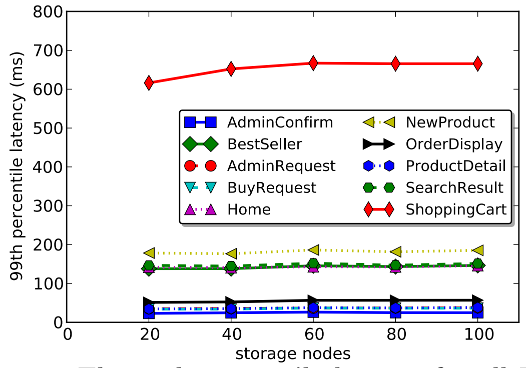

Figure 6: The 99th percentile latency for all TPC-W web interactions remains nearly constant as the number of machines increases

breakdown by comparing the time spent waiting for responses with the overall response time. The round trip latency to the storage nodes dominates query processing time, comprising on average 92% of total response time. In the worst case, local processing was responsible for 11% of latency, which is typical for short-running, distributed queries.

8.2.2 Latency of Parallel Refresh

The views used to answer queries such as BestSeller and AdminConfirm are updated periodically. Since we relax Invariant 2 to bound only the amount of work performed serially in each partition, instead of the total work required, these queries fall into SI-3. However, Figure 7 shows that since each partition can be processed in parallel, the overall time taken for the refresh step increases only slightly (less than a second) as we scale from 20 to 100 machines.

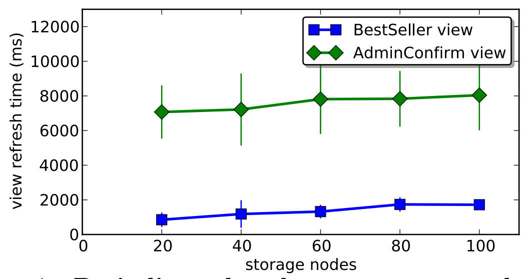

Figure 7: Periodic update latency stays nearly constant due to the requirement of bounded serial work.

9. RELATED WORK

Prior work on the automatic selection of materialized views, such as the static approach taken by Agrawal et al. [2]

or the dynamic approach taken by Kotidis and Rossopoulos [17], has focused on selecting views that will speed up a given workload. Generally the problem is framed as follows: Given a workload and a size bound, materialize the set of views that based on estimation will result in the greatest improvement in the performance of the system. Additionally, Gupta et al. [14] present algorithms that consider the cost of maintenance when selecting views. However, their techniques focus on minimizing the average cost of maintenance and query execution, not ensuring scale independence. In contrast, PIQL creates an IMV when it will fundamentally change the scalability of a query, not just when it could speed up execution. Thus, instead of just reducing response time, we ensure scale independence for an application before it is deployed to production. Additionally, our novel static analysis algorithms ensure that the size of the selected view can never grow faster than linearly relative to the base data.

DBToaster [3] also recursively creates materialized views to reduce the work of incremental maintenance . However, their system only works for views with aggregation and does not bound the total number of operations required.

Agrawal et al. [1] describe a large-scale declarative system that incrementally maintains materialized views. However, unlike PIQL, their query language does not support joins of more than two tables, unless they are all joined on the same attribute. Thus, PIQL's language extensions and recursive view creation allow the scale-independent maintenance of a wider range of views. Further expansion of scale independence may be possible. In particular better support for predicates with inequalities could be added through the inclusion of other data structures such as range trees [8].

10. CONCLUSION

The ability to anticipate and handle rapid growth is critical for modern large-scale interactive applications. Precomputation through the automatic creation of incrementally maintained materialized views is a powerful tool that can allow these sites to run more complex queries while still meeting their performance objectives. This technique shifts query processing work from execution time to insertion time, and can not only improve performance but also fundamentally change the scaling behavior of a wide range of queries.

In this paper, we presented a scale-independent view selection system, which not only recognizes such cases but also leverages novel static analysis techniques to ensure the automatically created views will not themselves become scaling bottlenecks. These algorithms ensure that created views will preserve invariants on both the storage required and the cost of maintenance as the scale of an application increases. Additionally, we described an annotation that allows PIQL to detect and mitigate common sources of workload hot spots. Together, these techniques allow PIQL to provide a relational engine that preserves the productivity benefits of SQL while maintaining the performance predictability of today's distributed storage systems.

Acknowledgments

We would like to thank Henry Cook, along with the reviewers, for feedback that significantly improved the content of this paper. This research is supported in part by NSF CISE Expeditions award CCF-1139158 and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, Blue Goji, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Quanta, Samsung, Splunk, VMware and Yahoo!.

11. REFERENCES

[1] P. Agrawal et al. Asynchronous view maintenance for vlsd databases. In *SIGMOD*, 2009.

[2] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, 2000.

[3] Y. Ahmad, O. Kennedy, et al. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *Proc. VLDB Endow.*, 5(10), 2012.

[4] M. Armbrust, K. Curtis, T. Kraska, A. Fox, M. J. Franklin, and D. A. Patterson. PIQL: Success-tolerant query processing in the cloud. *PVLDB*, 5(3), 2011.

[5] M. Armbrust et al. Scads: Scale-independent storage for social computing applications. In *CIDR*, 2009.

[6] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD*, 1986.

[7] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *VLDB*, 1991.

[8] M. Chaabouni et al. The point-range tree: a data structure for indexing intervals. In *Proc. of ACM CSC*, 1993.

[9] L. S. Colby et al. Algorithms for deferred view maintenance. *SIGMOD Rec.*, 25(2), 1996.

[10] E. Cunha et al. Analyzing the dynamic evolution of hashtags on twitter: a language-based approach. In *Workshop on Languages in Social Media*, 2011.

[11] G. DeCandia et al. Dynamo: amazon's highly available key-value store. *SIGOPS*, 41, 2007.

[12] J. Gray, A. Bosworth, A. Layman, D. Reichart, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. 1996.

[13] A. Gupta, D. Katiyar, and I. S. Mumick. Counting solutions to the view maintenance problem. In *Workshop on Deductive Databases, JICSLP*, 1992.

[14] H. Gupta and I. Mumick. Selection of views to materialize in a data warehouse. *Knowledge and Data Engineering, IEEE Transactions on*, 17(1), 2005.

[15] J. Kincaid. Zuckerberg: Online sharing is growing at an exponential rate. http://tinyurl.com/cskurl3.

[16] C. Koch. Incremental query evaluation in a ring of databases. In *PODS*, 2010.

[17] Y. Kotidis et al. Dynamat: a dynamic view management system for data warehouses. *SIGMOD Rec.*, 28(2), 1999.

[18] W. Labio et al. Performance issues in incremental warehouse maintenance. In *VLDB*, 2000.

[19] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *WWW*, 2005.

[20] G. Luo et al. Locking protocols for materialized aggregate join views. In *VLDB*, 2003.

[21] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. *SIGMOD Rec.*, 30(2), 2001.

[22] M. E. J. Newman. Power laws, pareto distributions and zipf's law. *Contemporary Physics*, 46, 2005.

[23] M. T. Özsu and P. Valduriez. *Principles of distributed database systems (2nd ed.)*. 1999.

[24] D. Quass and J. Widom. On-line warehouse view maintenance. In *SIGMOD*, 1997.

[25] K. Salem et al. How to roll a join: asynchronous incremental view maintenance. *SIGMOD Rec.*, 29(2), 2000.

[26] B. Trushkowsky et al. The scads director: scaling a distributed storage system under stringent performance requirements. In *FAST*, 2011.

[27] P. Valduriez. Join indices. *ACM Trans. Database Syst.*, 12(2), 1987.

[28] K. Weil. Measuring tweets. http://blog.twitter.com/2010/02/measuring-tweets.html.