A Session-Based Approach to Fast-But-Approximate Interactive Data Cube Exploration

NIRANJAN KAMAT, The Ohio State University
ARNAB NANDI, The Ohio State University

With the proliferation of large datasets, sampling has become pervasive in data analysis. Sampling has numerous benefits – from reducing the computation time and cost to increasing the scope of interactive analysis. A popular task in data science, well-suited towards sampling, is the computation of fast-but-approximate aggregations over sampled data. Aggregation is a foundational block of data analysis, with data cube being its primary construct. We observe that such aggregation queries are typically issued in an ad-hoc, interactive setting. In contrast to one-off queries, a typical query session consists of a series of quick queries, interspersed with the user inspecting the results and formulating the next query. The similarity between session queries opens up opportunities for reusing computation of not just query results, but also error estimates. Error estimates need to be provided alongside sampled results for the results to be meaningful. We propose *Sesame*, a rewrite and caching framework that accelerates the entire interactive session of aggregation queries over sampled data. We focus on two unique and computationally expensive aspects of this use case: query speculation in the presence of sampling, and error computation, and provide novel strategies for result and error reuse. We demonstrate that our approach outperforms conventional sampled aggregation techniques by at least an order of magnitude, without modifying the underlying database.

CCS Concepts:• **Information systems → Online analytical processing engines;**• **Human-centered computing →** *Activity centered design;*

Additional Key Words and Phrases: Error Reuse, Interactive Visualization, Faceted Exploration, Session, Aggregation

ACM Reference Format:
Niranjan Kamat, and Arnab Nandi, 2015. A Session-Based Approach to Fast-But-Approximate Interactive Data Cube Exploration *ACM Trans. Knowl. Discov. Data* 1, 1, Article 1 (January 2016), 25 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

With the rise of the internet era and data-driven disciplines, the amount of data collected is outpacing our ability to process it [Lohr 2012; Manyika et al. 2011]. Despite significant advances in modern hardware and database systems, the scale of the data is often a significant issue in the field of rapid data analysis. Modern systems often address this problem using parallelism through multithreaded or multi-core solutions [Kornacker and Erickson 2012]. However, despite such computational capabilities, analyzing large datasets to obtain valuable insights within interactive response times continues to be a challenge – studies [Liu and Heer 2014; Shneiderman 1984] have shown that surfacing results within the sub-second range significantly improves the analytics experience for the end-user.

In a wide variety of domains and use cases, users are happy to trade off accuracy for faster query response times via sampling – this is particularly applicable to domains that rely on data exploration. Sampling can also benefit accuracy-critical applications such as healthcare and finance during preliminary analysis, which can be followed by the exact computation. Though there have been several efforts towards improving query performance, there has been relatively little emphasis on bounded-time execution or online aggregation [Agarwal et al. 2013; Hellerstein et al. 1997]. Recently, there has been a renewed focus on approximate querying as a solution for interactive use cases [Babcock et al. 2003; Garofalakis et al. 2001; Rösch et al. 2013; Zeng et al. 2014].

This work is supported by the National Science Foundation, under grants 1453582 and 1422977.
Author's addresses: N. Kamat and A.Nandi, Computer Science and Engineering Department, The Ohio State University.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2016 ACM. 1539-9087/2016/01-ART1 $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

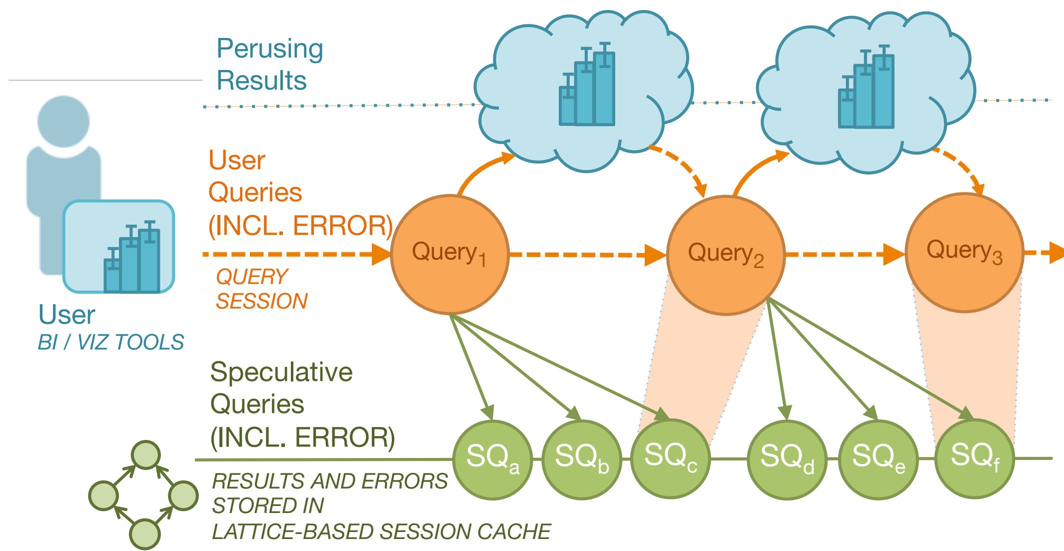


Fig. 1: Query Workload and Reuse Model: *Sesame* reuses result and error computations from previous queries of the session, and pre-populates the session cache with likely follow-up queries.

Sesame considers querying in an OLAP scenario, for which we use a **session-based query model**. In exploratory OLAP scenarios, we note that queries occur not in isolation, but as part of larger query *sessions* [Sapia 1999]. It is common for users to pose a query, and then issue follow-up queries based on the insights gleaned, or to query related parts of the data. Iterative, session-based queries also make up a significant fraction of the TPC-DS queries [Poess et al. 2007]. Hence, it is important to look at the query processing paradigm in the context of sessions. In such scenarios, since subsequent queries are dependent on the current user query, providing interactive response times carries a greater significance, as it helps the user iterate over the query session faster. Further, since each session query needs user supervision and constant refinement, the user's time is inherently coupled with the query response time. As we will see in the following sections, several opportunities arise from a session-based approach to sampled aggregation that can lead to a significant reduction in the response time.

With analytics playing an important role in data science, data aggregation has assumed an even greater importance. *Data cube*, which can be trivially explained as the union of all possible combinations of `GROUP BY` queries for different desired aggregate measures, is a fundamental and approximable construct of aggregation. A data cube is typically materialized offline – results for aggregate queries can then be obtained by running them over the cube. Cubes are often used by data analysts to better understand and explore the data. However, complete materialization of the cube is expensive and often untenable for large datasets, resulting in emergence of partially materialized cubes [Hanusse et al. 2011; Harinarayan et al. 1996]. Further, it is not possible to update the cube for all measures and scenarios, e.g., `MEDIAN`, or the `MIN` or the `MAX` tuple being deleted, etc. Offline materialization of the cube is also infeasible when a new measure is constructed by the user on the fly. Clearly, *online* data cube exploration methods can be considered more practical for such use-cases. In this age of Big Data-based analysis, avoiding computation of large cubes, while validating multiple hypotheses quickly in a query session, will be extremely useful.

The session-based approach to querying presents a unique opportunity in the context of online cube construction. As shown in Figure 1, while the user is perusing the results, the system can populate the session cache with likely follow-up queries, reducing the query response time. We use operators derived from the data cube model as the basis for query speculation. As described

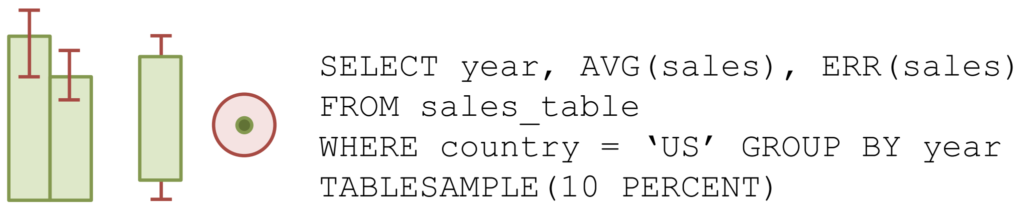


Fig. 2: Error estimates are an important component of sampled aggregation-based queries. `ERR()` represents error estimates such as standard deviation, standard error, or other variance-based computations that are expensive to compute.

in the following sections, we do not restrict reuse to simple prefetching or result caching [Battle et al. 2013; Kamat et al. 2014], but extend it to precomputation of expensive measures such as `VARIANCE`. As we will see in our experimental evaluation, predicting the user's behavior and pre-populating the session cache with both result and error estimates drastically reduces the average query response time. Error bars are essential in sampled aggregations. Errors for popular measures such as `SUM`, `MEAN`, etc. are typically evaluated using variance of the measure. Figure 2 depicts a sampled aggregation query, with some commonly-used error visualizations. In this example, as the user needs average sales for different years, reusing result will involve simply being able to use the values of average sales for different years from the cache. On the other hand, reusing errors will consist of using the computed variance of sales from the cache.

While variance is an essential component of error estimation, computing it can be expensive. Hence, in order to leverage the benefits of approximate querying more effectively, it is essential that we expedite variance computation. An intuitive way to do so is to reuse variance computation. *Sesame* is the first to focus on this critical aspect – in *DICE* [Kamat et al. 2014], queries with variance are run after the user query, burdening the system with an additional query, and significantly delaying knowledge of the error estimate to the user (detailed differences between *DICE* and *Sesame* are provided in Section 1.3). In *BlinkDB* [Agarwal et al. 2013], a popular offline sample computation system, the authors use *error profile* to approximate the sample variance, which is not only highly variable, but also introduces error in the error determination itself! It should be noted that *Sesame*'s contributions are towards error reuse and speculation strategies that focus on session-based sampled aggregation workloads, and are strictly orthogonal to various offline sampling techniques, such as those employed by BlinkDB. These systems can always be combined (with considerable engineering effort) to yield additional benefits: incremental samples used in BlinkDB can be mapped to our table shards, and their online sample selection strategy can be carefully integrated with ours. In contrast, our system leverages a novel lattice-based session cache to reuse variance computation, in addition to the results.

One way to expedite expensive computations is to parallelize them. A *distributive* measure is defined as *"an aggregate function that can be computed in a distributed way"*, and an *algebraic* measure is defined as *"an aggregate function that has a bounded number of arguments, each of which is obtained by applying a distributive function"* [Malinowski and Zimányi 2008]. Hence, the calculation of distributive and algebraic measures can be made highly parallelized as well as reusable, providing tremendous speedups. In this paper, we provide new strategies for cache and result reuse by leveraging previously unexplored properties of statistical measures, such as variance and standard error, which have not been classified as being algebraic so far, to answer sample-based queries. Reusing variance computation helps avoid processing the entire dataset to calculate variance. We can use result sets of already computed queries, which can be smaller by multiple orders of magnitude, to do so. The reduced data size of these results increases the likelihood of them fitting into various levels of the cache and memory hierarchy, further speeding up query execution. Further, in Section 4.3, we show how the error reuse framework extends to user-defined measures as well.

Considerations: There are three key challenges when building an architecture for session-oriented sampled aggregation queries. Our first challenge is, *how can we find out the best queries to speculatively execute?* Second, in terms of result reuse, *given a choice between different parts of the cache that can be used to answer a query, which ones should be chosen?* Third, *how can we provide exact and fast computation of variance within interactive response times?* As shown by our experimental evaluation, *Sesame* provides lower query response times and consistent performance both within a session as well as between query sessions, thus enabling a smooth interactive user experience.

1.1. Contributions

The overarching goal of *Sesame* is to accelerate sampled aggregation queries without modifying the underlying database by providing a middleware. As shown in our evaluation section, *Sesame* achieves speedups of up to $\mathbf{25}\times$ using a single-node backend (PostgreSQL) and up to $\mathbf{4}\times$ using a distributed backend (Impala). This is made possible with the following contributions:

- *Query Speculation-Based Sampling:* We provide execution and reuse strategies that are aware of sharding, approximation, and the session-based OLAP workload, to enable speculative execution in the context of sampling.
- *Cube Lattice-based Session Cache:* We provide a cube lattice-based session cache that expedites the search of reusable results, and helps with the cube region-aware allocation of caching resources.
- *Error Reuse:* We show that error reuse is an often overlooked and yet expensive component of sampled aggregation queries, and describe methods to extend reusability to variance computation.
- *Speculating on Errors, not just Results:* We note that variance is an algebraic measure, and speculate on not just the results, but also the error.

1.2. Paper Organization

We now discuss the organization of the rest of the paper. Section 2 discusses the faceted query model used in *Sesame*. Section 3 presents the overall *Sesame* system. Section 4 discusses our efforts towards extending result reuse from simply measures to errors as well. Section 5 presents our experiments, which are based on a completely new, and much faster, backend implementation. They illustrate the potential of query speedups better – achieving up to $25\times$ speedups, as opposed to 33% provided by *DICE*. Finally, we present the related work in Section 6, our limitations in Section 7, and the conclusion and ideas for future work in Section 8.

1.3. Comparison with *DICE*

Sesame is an extension of our previous work, *DICE* [Kamat et al. 2014]. The faceted query model (Section 2) introduced in *DICE* has been extended to consider multiple dimensions in the `GROUP BY` clause by *Sesame*. A new operator *Resample* is also introduced. We have repeated the description of *Accuracy Gain* (Section 3.1) for completeness, and have elucidated the rationale behind its design better. Our cache design (Section 3.2.1), selection of optimal views for reuse (Section 3.2.2), and the lookup algorithm (Section 3.2.3) are novel as well. The overall algorithm (Section 3.3) draws inspiration from *DICE*, while having better individual modules. Our extension of reuse from only considering measures to errors as well is also a novel concept (Section 4). Our experiments (Section 5) are based on an entirely new engine. While the description of table shards has been repeated from *DICE*, we have now provided reasons behind their randomness (Appendix A).

2. QUERY MODEL

We use the data cube querying techniques to model user queries since they provide the ideal framework for aggregate querying. We note that *Sesame* has been designed for execution of `SELECT`, `PROJECT`, and `SAMPLE` queries. We do not consider `JOIN` queries as the resulting increase in the number of speculative queries would greatly diminish speculation efficiency. While faceted traver-

sals have been introduced in *DICE*, we present them here for completeness. We have also improved their exposition. We also introduce a new operator called *Resample*.

As described in Section 1, *Sesame* uses a **session-based query model** due to queries occurring in sessions in OLAP scenarios. For any aggregation in the data cube, the number of possible follow-up queries is extremely large (number of groups in the cube $-$ 1). This creates a challenge for our speculative execution techniques – which queries do we pick? Hence, we present a model where the user explores the cube in a series of *faceted traversals*, with the subsequent query differing from the current query on a single dimension, thereby restricting the space of queries in a principled fashion. The *faceted cube exploration* model is used to guide user exploration at the frontend, and also to speculate and execute a subset of the next possible queries at different sampling rates to maximize the probability of the next query being present in the query cache at a high sampling rate.

In the context of cube exploration, the definitions of *cube*, *region*, and *group* are as per the original data cube paper [Gray et al. 1997]. A region denotes a node in the cube lattice (Figures 3 and 5), and a group denotes tuples with the same values for attributes of that region. In the rest of the paper, we use the following schema for illustrative purposes: table *events* catalogs all informative system events across the cluster, and has two hierarchical dimensions and a measure – *location [zone:datacenter:rack], time [month:week:hour], and iops.*

We introduce the term *facet* as the basic state of exploration of a data cube, drawing from the use of category counts in the exploratory search paradigm of faceted search [Tunkelang 2009]. Facets are meant to be perused in an interactive fashion – a user is expected to fluidly explore the entire data cube by successively perusing multiple facets. Intuitively, a user explores a cube by inspecting a *facet* of a particular region in the data cube – a histogram view of a subset of groups from a region. The user then explores the cube by traversing from that facet to another. This successive facet can be a **parent** facet in the case of a roll-up, a **child** facet in the case of a drill-down, a **sibling** facet in the case of change of a group value, or a **pivot** facet in the case of a change in the inspected dimension. Thus, the user is effectively moving around the cube lattice to either a parent region, or a child region, or remaining in the same region using sibling and pivot traversals to look at the data differently. The formal definitions and examples are given below.

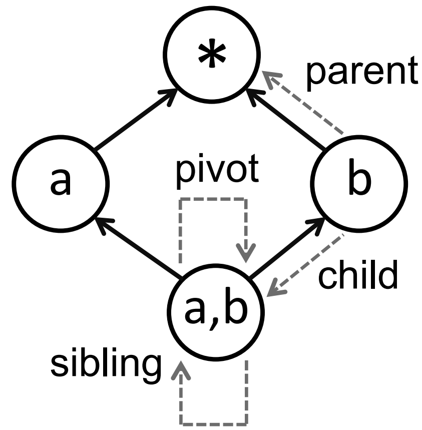


Fig. 3: Faceted Traversals in a Cube Lattice.

Facet: For a region r in cube C, a facet f is a set of groups $g \in r(d_{1...n})$, such that the group labels can differ on multiple common dimensions d_i, i.e. $\forall \overrightarrow{g_a}, \overrightarrow{g_b} \in f, d_i(\overrightarrow{g_a}) \neq d_i(\overrightarrow{g_b}) \wedge d_j(\overrightarrow{g_a}) = d_j(\overrightarrow{g_b})$, where $i \neq j$, and d_i are the *grouping dimensions*, and d_j are the *bound dimensions*. In its SQL representation, a facet in a region contains a `GROUP BY` on the grouping dimensions, and a conjunction of `WHERE` clauses on the bound dimensions. A facet can be referred to using the notation $f(\overrightarrow{d_g}, \overrightarrow{d_b : v_b})$, where $\overrightarrow{d_g} \cup \overrightarrow{d_b}$ denotes the dimensions in the region, $\overrightarrow{d_g}$ denotes the grouping dimensions, and $\overrightarrow{d_b : v_b}$ denotes the bound dimensions and their corresponding values. Thus, the measure `COUNT` on the dimension *iops* along with the facet $f(zone, month : m_1, week : w_1)$ gives a histogram of I/O failure counts grouped by different zones for a specific month and week. In our notation, $zone$ represents the grouping dimension $\overrightarrow{d_g}$, and $month : m_1, week : w_1$ represents the bound dimensions $\overrightarrow{d_b : v_b}$.

Facet Session: A facet session $\vec{F}$ is an ordered list of facets $f_{1...n}$ that a user visits to explore the data cube. A traversal can be defined by a transition from one facet to another. We now define and

provide examples for our four traversals – *Parent*, *Child*, *Sibling*, and *Pivot* – which draw inspiration from similar traversals over a data cube. All traversal examples given below are with respect to the current facet $f(zone, month : m_1, week : w_1)$.

Parent Facet: A parent facet is defined as a facet obtained by generalizing any of the bound dimensions, while keeping grouping dimensions the same. Thus, a facet $f_p(\overrightarrow{d_{pg}}, \overrightarrow{d_{pb} : v_{pb}})$ is a parent of the facet $f(\overrightarrow{d_g}, \overrightarrow{d_b : v_b})$ if $\overrightarrow{d_{pg}} = \overrightarrow{d_g}$ and $\overrightarrow{d_{pb} : v_{pb}}$ represents a parent group of $\overrightarrow{d_b : v_b}$. The parent facet $f(zone, month : m_1)$ generalizes the dimension *time* from the prior example. In the *time* hierarchy, the lower bound $week : w_1$ has been removed from the bound dimensions $month : m_1, week : w_1$.

Child Facet: A child facet is defined as a facet obtained by specializing any of the bound dimensions. Thus, a facet $f_c(\overrightarrow{d_{cg}}, \overrightarrow{d_{cb} : v_{cb}})$ is a child of the facet $f(\overrightarrow{d_g}, \overrightarrow{d_b : v_b})$ if $\overrightarrow{d_{cg}} = \overrightarrow{d_g}$ and $\overrightarrow{d_{cb} : v_{cb}}$ represents a child group of $\overrightarrow{d_b : v_b}$. The child facet $f(zone, month : m_1, week : w_1, hour : h_1)$ specializes the bound *time* dimension, $month : m_1, week : w_1$, by appending $hour : h_1$ to it.

Sibling Facet: A sibling facet is defined as a facet obtained by changing the value of exactly one of the bound dimensions. Thus, a facet $f_s(\overrightarrow{d_{sg}}, \overrightarrow{d_{sb} : v_{sb}})$ is a sibling of the facet $f(\overrightarrow{d_g}, \overrightarrow{d_b : v_b})$ if $\overrightarrow{d_{sg}} = \overrightarrow{d_g}$, $\overrightarrow{d_{sb}} = \overrightarrow{d_b}$ and $\overrightarrow{v_{sb}}$ and $\overrightarrow{v_b}$ differ on exactly one value. The sibling facet $f(zone, month : m_1, week : w_2)$ changes the value of *week* in the facet $f(zone, month : m_1, week : w_1)$ from w_1 to w_2.

Pivot Facet: A pivot facet, which is slightly more complex than the above three, is defined as any facet obtained by interchanging a grouping dimension with a bound dimension. Thus, a facet $f(\overrightarrow{d_g}, \overrightarrow{d_b : v_b})$ can be pivoted to the facet $f_v(\overrightarrow{d_{vg}}, \overrightarrow{d_{vb} : v_{vb}})$ if $\overrightarrow{d_{vg}} \in \overrightarrow{d_b} \wedge \overrightarrow{d_g} \in \overrightarrow{d_{vb}}$ and $\overrightarrow{d_b : v_b}$ and $\overrightarrow{d_{vb} : v_{vb}}$ have all but one bound dimension and value in common. The current facet $f(zone, month : m_1, week : w_1)$ can be pivoted over $zone : z_1$ giving us a pivot facet $f(week, zone : z_1, month : m_1)$, where we group over the dimension $week$ and bound the dimension $zone$ to z_1.

Resample: The above facets represent changes to the query. Resample operator, on the other hand, is used when the user is not satisfied with the result error level at the current sampling rate, and wishes to improve the result precision. It runs the user query at the additional sampling rate and combines the result with the already computed one. Note that this operator is only applicable for distributive and algebraic measures.

3. THE *SESAME* FRAMEWORK

Sesame presents a standard query execution workflow to the user, while using a set of several complementary techniques. Figure 4 presents an architectural outline of the system. Once a user query is received from the interactive query session, it is executed and the results are returned. After finishing the execution of the user query, *Sesame* calls the *speculation* module to enumerate the different follow-up queries, and executes an optimal subset of them. When possible, results and error computations are reused from the novel lattice-based session cache, which is designed for cube-based querying. The session is highly interactive, targeting a near-sub-second response time per query. This necessitates an in-memory-only architecture, which *Sesame* employs. *Sesame* connects to a SQL-capable database as its backend to execute the queries, and uses query rewriting and post-aggregation to coordinate reuse. *Sesame* uses data sharding, which is a pervasive and pragmatic technique for large data platforms.

We now give a brief overview of this section. First, we provide methods to generate the optimal set of speculative queries (Section 3.1). Next, we describe our cache structure and techniques to look it up (Section 3.2). Finally, we present our overall algorithm for speculative query execution and result reuse (Section 3.3).

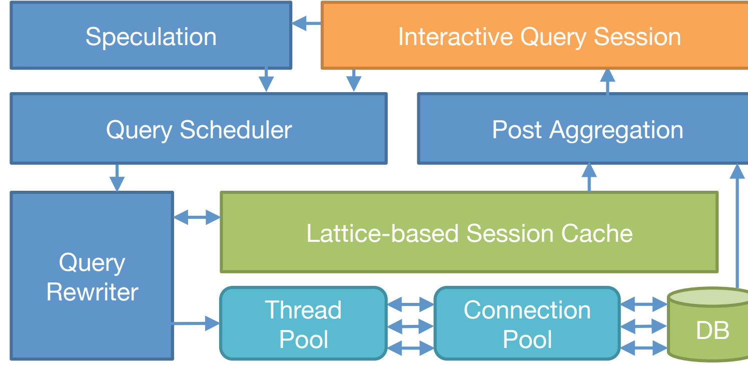


Fig. 4: *Sesame* System Architecture.

3.1. Session-aware Speculation Schedule

It is clear from the faceted model that each ad-hoc query can yield a significantly large number of speculative queries. Given that the time available for speculative execution is bounded, it is typically not possible to execute all speculative queries over the entire data. It is necessary to prioritize execution of speculative queries to maximize the likelihood of results for the next query being returned from the cache.

Accuracy Gain Heuristic: Accuracy Gain was used in *DICE* for determining *goodness* of a speculative query. We repeat it here with better exposition. In order to schedule speculative queries at different sampling rates, we need to know the reduction in sampling error at different rates. However, it cannot be known before actually running the query. As the standard deviation and consequently, the error reduces as $O(\frac{1}{\sqrt{n}})$ for commonly-used aggregate functions, where n is the sample size, in a similar fashion as [Agarwal et al. 2013] who use it build their *error profile*, we use the following heuristic to estimate the increase in accuracy due to a unit increase in sampling rate.

$$AccuracyGain(R_{curr}) = c * (\frac{1}{\sqrt{R_{curr}}} - \frac{1}{\sqrt{R_{curr}+1}}) \quad (1)$$

where R_{curr} is the current sampling rate and c is the proportionality constant. We also give lesser importance to queries that have been recently issued by the user – although this is intuitive, we have been unable to determine its empirical benefit, and can consequently be excluded. We model selection of a subset of speculative queries to execute, given the queries executed in the session so far, as the following *Mixed Integer Linear Programming* problem.

MAXIMIZE:
$$\sum_{q \in Q} Prob(q) \cdot AccuracyGain(SR) \cdot Recency(q) \cdot x_q$$
CONSTRAINT:
$$\sum_{q \in Q} ExecutionTime(q) \cdot x_q \leqslant SpeculationTime$$
GIVEN: $x_q \in \{0, 1\}$

where q is a speculative query amongst the set of all speculative queries Q (all possible follow-up queries according to the traversal model given in Section 2) at every possible sampling rate *SR* where $0 \leqslant SR \leqslant Number\ of\ Shards$, $ExecutionTime(q)$ is the estimated time taken to execute query q, and x_q indicates whether the query q is part of the selected queries. $Prob(q)$ gives the probability of a query q, which should be obtained from the query logs, $ExecutionTime(q)$ is the estimated execution time, and $SpeculationTime$ is the expected total speculation time, which is estimated

using the user behavior in the query session thus far. We have used the *memory model* [Averell and Heathcote 2011] in constructing the recency function, $Recency(q)$, although other functions might also be useful. We have not provided its detailed description for conciseness as we could not empirically demonstrate its utility. The individual components of the metric are normalized and multiplied with each other as is common-place in metric design, and since our experience in using them guides us towards it as well. The optimal set of speculative queries is determined using the Gurobi Optimization Solver[1].

3.2. Cache Lookup

We now present the different components involved in retrieving results from the cache. We show how the query cache can be modeled as a lattice, and the benefits of doing so (Section 3.2.1). We provide techniques to determine which cached view amongst different reusable views should be used to retrieve results for a user query, in order to minimize the execution time (Section 3.2.2 and Section 3.2.3).

3.2.1. Lattice-based Session Cache. We use a novel, yet intuitive cache structure based on materialized views in the backend database. We model the cache as a lattice, mimicking the data cube model, with each cube region having a cache associated with it. This gives us a fine-grained control and greater flexibility over caching resources – caches near the current query region can dynamically be made larger than those further away. The lattice structure of the cache also helps speed up the lookup for potential views that can be reused to answer the user query, by reducing the size of the cache that needs to be searched, as described in Section 3.2.3. This structure is intuitive and more suitable for data cube-based querying, as opposed to the commonly-occurring flat cache structure.

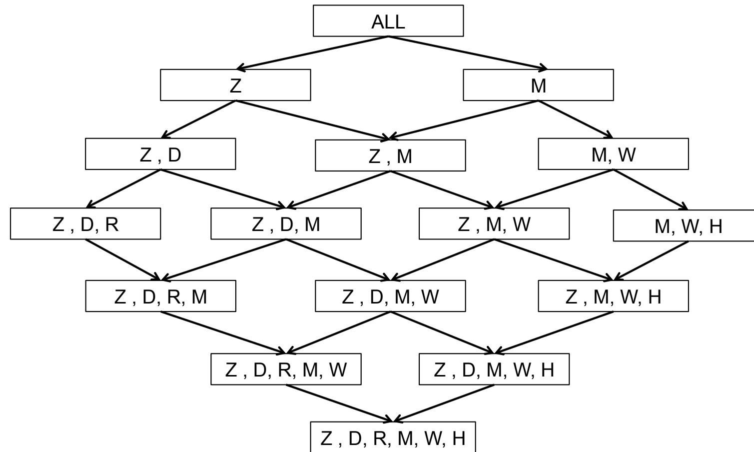


Fig. 5: Lattice-based Cache: This figure represents the lattice structure of the cube for the schema presented in Section 2 consisting of two hierarchies, location and time, and measure iops. *ALL* stands for the top-most region, without any `GROUP BY` clause. Other regions are denoted by the initial letter of their dimensions. We model the cache in a similar fashion as the lattice (Section 3.2.1). A query can be answered only from its descendant regions – reducing the cache size that needs to be searched, and allowing us to look through caches belonging to regions having smaller view sizes first (Sections 3.2.2 and 3.2.3).

[1] www.gurobi.com

3.2.2. Optimal View to Reuse. For distributive and algebraic measures, it is clear that the query result can be obtained in its entirety by reusing the result of any one of its child queries. We discuss different scenarios that provide opportunities for result reuse, and suggest the best decision in each case. Due to the usage of memory-resident data and the lack of indexes, we can safely assume that the optimal view to reuse would be the one with the least number of tuples [Harinarayan et al. 1996], as queries run on larger views will take longer to scan through. To the best of our knowledge, this is the first piece of work to enumerate the different options and provide the optimal solution in each case. We also present an extension in the case that the data is horizontally partitioned into shards. Similar to [Harinarayan et al. 1996], we denote by $Q_1 \preceq Q_2$ if Q_1 can be answered in its entirety by Q_2.

Rule 1: Consider the case where $Q_1 \preceq Q_2 \preceq Q_3$, and we have to choose between Q_2 and Q_3 to answer Q_1. In this case, Q_2 will have lesser number of tuples than Q_3, and should be preferred. For example, views constructed through `GROUP BY zone, datacenter` and `GROUP BY zone, datacenter, rack` can be used to answer queries having `GROUP BY zone`, with other parts of the queries being the same. However, `GROUP BY zone, datacenter` will be smaller than `GROUP BY zone, datacenter, rack`, and should be preferred.

Rule 2: Suppose $Q_1 \preceq Q_2$, $Q_1 \preceq Q_3$, $Q_2 \npreceq Q_3$ and $Q_3 \npreceq Q_2$. In this case, there is no clear choice between using Q_2 or Q_3 to answer Q_1, based on their relationship alone. In this case, we choose the view with lesser number of tuples. For example, it is unclear whether views as a result of `GROUP BY zone, datacenter` or `GROUP BY zone, month` will be smaller.

Rule 3 (Sharding Effect): Consider the case where two views were the result of the same query being run on two different shards, and we had to choose between them to answer the user query. In this case, we again choose the view with fewer number of tuples. In the case of data skew, this is an important case to consider.

3.2.3. Lookup Algorithm. The principles given above, governed by choosing views having the smallest possible size, are used in designing the view lookup algorithm. In order to satisfy a user query, we look up views in the cache that can completely answer the user query. The lattice structure of the cache helps in this process since only views in the region that the query belongs to or its descendant regions can be used to answer a given query. Thus, we can avoid searching through a large section of the query cache. The lookup algorithm for potential superset views can then be given as follows.

1. First, we look up in the cache of the region that the query lies in, and add the usable views to a candidate list.

2. Whenever all views belonging to a region are used, we add the usable views from its child regions to the candidate list (Scenario 2).

3. Step 2 is recursed till the desired sampling rate is met, or all the caches of the descendant regions have been searched.

4. If the sampling rate is not met, we run the user query on the shards that have not been used so far.

We provide a brief example of an application of the above algorithm, where we model the behavior of a network analyst as he tries to understand the measure *iops* across various server locations over time. Suppose the current user query is:

```
SELECT month, AVG(iops) FROM events
WHERE zone = 'NorthWest' GROUP BY month
```

First, we check whether the query is present in the cache of the region $\{zone, month\}$ or whether superset views are present in that cache. If the number of views present in the cache is insufficient, we look at the cache of its child regions, $\{zone, datacenter, month\}$ and $\{zone, month, week\}$, and add the usable views to the list. If all views belonging to either of these regions have been used, the views belonging to its child regions are added, and the process followed recursively. Selecting views along this hierarchy is straight-forward using the lattice cache presented in Figure 5.

3.3. The *Sesame* Algorithm

In this section, we summarize all the design decisions involved in building *Sesame*, and present the overall algorithm.

ALGORITHM 1: SesameSession

```
1  while UserQuery do
2      Views = FindReusableViews(UserQuery)
3      RunQueries(UserQuery, Views)
4      if Count(Views) < SamplingRate then
5          CountOfExtraShardsNeeded = SamplingRate - Count(Views)
6          Shards = FindUnusedShards()
7          RunQueries(UserQuery, Shards)
8      end
9      SpecQueries = FindOptimalSet(UserQuery)
10     RunQueries(SpecQueries)
11 end
```

Algorithm 1 describes the overall flow of *Sesame*. First, the reusable views are found (Line 2). The user query is then run on these views (Line 3). Based on the number of extra shards needed to meet the sampling requirement (Lines 4-5), we find the shards on which the query needs to be run (Line 6), and run them (Line 7). The optimal set of speculative queries is then determined (Line 9), and executed till the user issues his next query (Line 10).

ALGORITHM 2: FindReusableViews

```
Input: Query
Output: Views
1  Views = FindExactMatchInCache(Query)
   /* Find optimal views in the query region if the sampling rate has not
      been met.                                                          */
2  Views = Views ∪ FindOptimalViews(Query, Region)
   /* Find optimal views recursively in the child regions till the sampling
      rate is met.                                                       */
3  Views = Views ∪ FindOptimalViews(Query, Descendants)
4  return Views
```

Algorithm 2 articulates the steps involved in finding the reusable views. First, we check if there are any cached results for the exact query (Line 1). If the sampling rate is not met, we find the optimal set of views to reuse in the query region as explained in Section 3.2.2 (Line 2). Next, we check amongst its child regions hierarchically (Line 3), choosing optimal views at each step.

4. ERROR COMPUTATION IN *SESAME*

As Section 1 illustrates, variance is a necessary component of sampled aggregation queries, and can be more expensive to compute than measures such as SUM, MEAN, COUNT, etc. Variance computation can be quadratic in complexity with respect to the number of tuples. In a pre-experiment on a standard database system using a TPC-DS derived dataset and a real-user workload, **we observed that including variance calculations increases query response time by nearly 54.6%**, when compared with simply including the other measures given above. (We provide further details in Appendix B). Thus, including variance can be substantially more expensive, and can lower the query response time benefit of sampling.

A common approach to expediting cube materialization and online cube computation is to distribute the effort or reuse prior computations for efficiency [Nandi et al. 2012; Ng et al. 2001]. In

Sesame, we build upon this concept in the following ways. Beyond just results, we use the fact that variance is an algebraic measure, and that results from variance computation can be reused. We compare various techniques that can be used to calculate variance, and detail which ones are viable for reusing results, and which ones yield faster execution times (Sections 4.1 and 4.2).

4.1. Error Reuse

There has been extensive work in reusing results for measures such as SUM, MEAN, etc. [Chirkova and Yang 2011; Halevy 2001]. However, *Sesame* is the first system that provides a framework for reusing the *errors* as well. A common representation of variance calculation is:

$$v = \frac{1}{n-1}\Big(\sum_{i=1}^{|G|} n_i(m_i - m)^2 + \sum_{i=1}^{|G|}(n_i - 1)v_i\Big) \tag{2}$$

which we call $Variance_{Common}$, where v is the combined variance of all groups, n is the number of tuples in the sample, m is the mean of the sample, v_i is the variance of group i, n_i is the number of tuples in group i, m_i is the mean of group i, and $|G|$ is the number of groups. This representation of variance lets it be utilized as an algebraic measure as a function of mean, count, and variance of individual groups. In this fashion, we can obtain the overall variance by combining the results from groups, without having to access the underlying data in the variance combination step.

We now look at another representation of variance [Welford 1962]:

$$v = \frac{1}{(n)(n-1)}\left(n\sum_{i=1}^{n} x_i^2 - \Big(\sum_{i=1}^{n} x_i\Big)^2\right) \tag{3}$$

which we call $Variance_{Fast}$. Here, x_i items are present in a sample of size n. We can see that by keeping a counter for $\sum_{i=1}^{n} x_i$ and $\sum_{i=1}^{n} x_i^2$, variance can be easily computed, and thus, this representation is algebraic as well.

It is worth noting the differences between $Variance_{Common}$ and $Variance_{Fast}$ from the perspective of interactive querying. In $Variance_{Common}$, the values for the mean, count, and variance for each group need to be stored, whereas in $Variance_{Fast}$, we only need to keep track of $\sum_{i=1}^{n} x_i$ and $\sum_{i=1}^{n} x_i^2$. Our experiments show that the second technique is much faster (Section 5.3.6). We further elaborate on the underlying reasons in Section 4.2.

Another famous representation of variance developed by Welford et al. [Welford 1962] and presented by Knuth [Knuth 2014] computes the running values for mean and variance as follows:

$$\mu_n = \mu_{n-1} + \frac{x_n - \mu_{n-1}}{n} \tag{4}$$

$$v_n = v_{n-1} + (x_n - \mu_{n-1}) \cdot (x_n - \mu_n) \tag{5}$$

which we call as $Variance_{Incremental}$. Here, μ_n is the mean of n items, μ_{n-1} is the mean of the first $n-1$ items, x_n is the nth item, v_n is the variance of n items, and v_{n-1} is the variance of the first $n-1$ items. We can see that since it incorporates a single value at every step, it is not conducive for large-scale reuse.

Another variance representation derives from Equations 4 and 5, and presents a method for combining means and variances of 2 groups [Chan et al. 1982]:

$$\mu = \mu_1 + n_2 \cdot \frac{\mu_2 - \mu_1}{n_1 + n_2} \tag{6}$$

$$v_n = v_{n_1} + v_{n_2} + n_1 \cdot n_2 \cdot \frac{(\mu_2 - \mu_1)^2}{n_1 + n_2} \tag{7}$$

which we call $Variance_{BiCombination}$. However, this representation is not amenable to being evaluated using a SQL query, since it only combines two groups at a time. Thus, the only two representations that can be used for variance reuse are those given by $Variance_{Common}$ and $Variance_{Fast}$.

4.2. Efficiencies of Different Viable Variance Formulations

The main reason behind $Variance_{Fast}$ providing results much faster than $Variance_{Common}$ is that it does not involve variance calculation until the final step, and only needs to keep track of $\sum_{i=1}^{n} x_i$, and $\sum_{i=1}^{n} x_i^2$. $Variance_{Common}$ on the other hand determines the variance, mean, and count at every step for every group. Another reason is the lack of support for nested aggregate queries in standard databases such as PostgreSQL, MySQL, etc., which further degrades its performance. Hence, from a perspective of interactive approximate query processing, $Variance_{Fast}$ should be preferred over $Variance_{Common}$.

4.3. Extensibility to Other Measures

So far, we have seen how result reuse can be extended to variance. Due to the algebraic properties of variance and thereby of standard deviation, standard error and co-efficient of variation can be termed as algebraic as well, allowing us to reuse their computations. The error reuse and caching framework of *Sesame* can also be extended to any user-defined measure whose variance can be expressed in a closed form as a function of the variance of one of the measure dimensions. For example, for a user-defined measure given by $a * \texttt{AVG}(agg_1) + b$, where a and b are constants and agg_1 is a measure dimension, the variance of the measure can be given as $a^2 * \texttt{VARIANCE}(agg_1)$.

On the other hand, obtaining a closed form solution to the variance of *holistic* measures is not always possible. As a workaround, bootstrapping is a popular choice for variance estimation. However, for a subset of holistic measures known as partially algebraic measures [Nandi et al. 2012], it can be possible to compute the variance in an algebraic fashion. For example, for the partially algebraic measure `COUNT DISTINCT`, variance can be computed using the final measure value.

5. EXPERIMENTS AND EVALUATION

5.1. Experimental Setup

Sesame has been implemented in Java 7, and uses PostgreSQL 9.3 as the default database via JDBC protocol. It runs on an Ubuntu Linux 14.04.1 LTS system with a 24-core 2.4GHz Intel Xeon CPU, 256GB DDR3 @ 1866 MHz memory, and a 500GB @ 7200 RPM disk. Experiments were carried out in an in-memory-only setting. Buffers and caches at all levels were flushed before each experiment, and to avoid warm-up effects, results from the first of the 4 iterations were discarded, and the mean and the standard deviation across the remaining 3 iterations are reported.

Dataset and Query Workloads: Data and query workloads were determined using TPC-DS [Poess et al. 2007], since it is designed for evaluating performance of data warehousing queries. TPC-DS contains four iterative query sessions that we use to design a single query session, $Workload_{TPCDS}$, consisting of 35 queries. We also collected a **real-user workload** from a user study of five graduate students who were asked to analyze the TPC-DS dataset using the industry-standard *Tableau* data analytics tool[2]. Query logs from this usage were used to generate another query session ($Workload_{Real}$), comprising of 21 queries. Unless otherwise specified, the experiments were performed using $Workload_{TPCDS}$. *Sesame* achieved similar performance over smaller sessions as well. The data of size 20.7 GB was partitioned horizontally into shards, with each shard having $100,000$ tuples, which resulted in a total of 768 shards.

5.2. Metrics:

We use the metrics listed below to evaluate the different aspects of *Sesame*.

Average Response Time: We present the mean and the standard deviation of the query response time across the three runs. *(Error bars are shown, but may not be visible as the coefficient of variation is under 3% for all measurements – further explanation in Section 5.4.)*

[2]http://tableau.com

Cache Hit Rate: The *Cache Hit Rate* is defined as the fraction of the user queries that are answered from the query cache. It gives a sense of the effectiveness of query prediction.

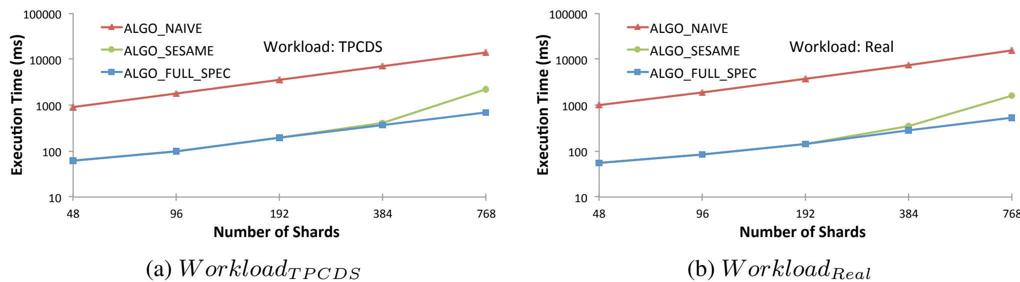
Speculation Hit Rate: Speculation Hit Rate is defined as the ratio of speculative queries that are reused by the user query to the total number of speculative queries run.

Overall Speculation Benefit: Running speculative queries results in the user query speeding up at an additional expense. Hence, a metric that takes both these factors into consideration to give us an overall picture about the benefits of speculation can be useful. We define the metric *Overall Speculation Benefit* as *Query Speedup* × *Speculation Hit Rate*, where *Query Speedup* is defined as the ratio of the execution time in the naive case (no speculation) to that using *Sesame*. The speculation benefit can be considered to outweigh its cost when this metric is greater than 1.

5.3. Results

We define $ALGO_{FULL_SPEC}$ as the algorithm without any limit on speculation duration. As a result, all user queries can be answered from the cache. The algorithm in which a subset of speculative queries is determined, as given in Section 3.1, and run within a bounded time interval is defined as $ALGO_{SESAME}$. Finally, we define $ALGO_{NAIVE}$ as the algorithm where speculative query execution is turned off. Thus, comparing $ALGO_{FULL_SPEC}$ to $ALGO_{NAIVE}$ gives us an idea about the benefits of speculation. Comparing $ALGO_{FULL_SPEC}$ to $ALGO_{SESAME}$ informs us about the detrimental effects of not answering a query from the query cache. The default maximum speculation duration was set to 30 seconds. Note that in the graphs, the number of shards is increased at an exponential rate. Due to the fact that the data is horizontally partitioned into shards, once the queries are run on individual shards, the results can be combined to present a single result. This is an expensive step and common to all the three algorithms described above. Since data aggregation is not the focus of this paper, and we want to better illustrate the benefits of speculative execution, the results containing data aggregation are presented only in Sections 5.3.7 and 5.3.8.

5.3.1. Response Time across Varying Data Sizes. We can note that the execution times for both $ALGO_{NAIVE}$ and $ALGO_{FULL_SPEC}$ increase linearly with increasing dataset size for both workloads (Figures 6a, 6b). **The benefit of our system is evident: $ALGO_{SESAME}$ is typically at least an order of magnitude faster than traditional database querying. As an anecdotal example, with 192 shards, $ALGO_{SESAME}$ is $18\times$ faster than traditional execution for $Workload_{TPCDS}$ and $25\times$ faster for $Workload_{Real}$.**



(a) $Workload_{TPCDS}$ (b) $Workload_{Real}$

Fig. 6: Execution Time for Different Dataset Sizes.

As expected, execution time for $ALGO_{SESAME}$ increases linearly until some of the queries need to be run on the underlying data instead of on the cached materialized views, as shown by the time differential between $ALGO_{FULL_SPEC}$ and $ALGO_{SESAME}$ for 384 and 768 shards. To illustrate the importance of retrieving results from the cached materialized views, we look at a couple of illustrative examples – even when 99% of the queries were answered from the cache (in the case of

384 shards), the speedup falls from 25.77 (for 100% *Cache Hit Rate*) to 21.35, and further to 9.45 when the *Cache Hit Rate* was 91% (in the case of 768 shards), for $Workload_{Real}$ (Figure 6b).

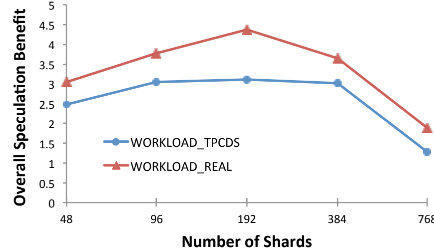
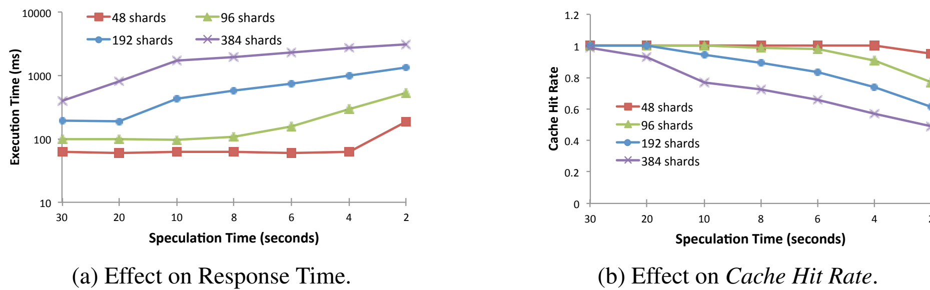


Fig. 7: Overall Speculation Benefit.

5.3.2. Overall Speculation Benefit. The *Overall Speculation Benefit* is similar to the query speedup initially, which increases as a greater fraction of time is spent actually running the query compared to the time spent in setting up the query execution, whose increase is relatively slower. Notably, the *Overall Speculation Benefit* is consistently greater than 1. As the sampling rate increases, a smaller fraction of the queries can be answered from the cache, causing the metric to decrease. Thus, this metric can be used to evaluate whether speculation is worthwhile.

5.3.3. Response Time across Varying Speculation Durations. As the speculation duration decreases beyond a threshold, execution time starts increasing (Figure 8a). Until this threshold is reached, user queries can be answered from the cache. Beyond this point, a greater fraction of the query will need to be run on the underlying data as opposed to the materialized views, which can be orders of magnitude more expensive.



(a) Effect on Response Time.

(b) Effect on *Cache Hit Rate*.

Fig. 8: Effect of Speculation Duration.

5.3.4. Cache Hit Rate across Varying Speculation Durations. As the speculation duration decreases beyond the above threshold, the *Cache Hit Rate* also correspondingly decreases linearly (Figure 8b). The linear decrease is expected since the number of queries that can be speculatively run will decrease linearly as well, thereby resulting in a corresponding drop in the *Cache Hit Rate*.

5.3.5. Comparison of View Sizes Needed. Since *Sesame* requires materializing additional speculative views, we investigate the space costs associated with both $ALGO_{NAIVE}$ and $ALGO_{SESAME}$ (Figure 9). Note that both techniques need to materialize the views after running the user query on individual shards in order to aggregate the results into a single result set. We can see that $ALGO_{SESAME}$ requires a maximum of six times the space of $ALGO_{NAIVE}$. Considering the benefits of improved response time and the low, plummeting costs of storage, this added expense is clearly worth the query speedups in the context of interactive execution.

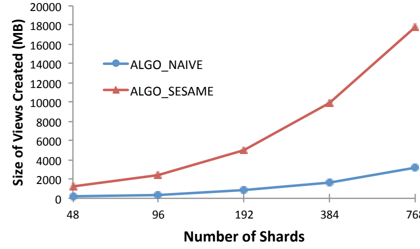


Fig. 9: Comparison of View Sizes Needed.

5.3.6. Comparison of $Variance_{Fast}$ and $Variance_{Common}$. In Section 4.2, we have discussed the reasons behind $Variance_{Fast}$ providing results faster than $Variance_{Common}$. With increasing data size, a greater fraction of the queries needs to be computed by accessing the underlying data, causing the benefit of using a faster variance calculation technique to become more prominent, as shown by the increase in speedup for 768 shards (Figure 10).

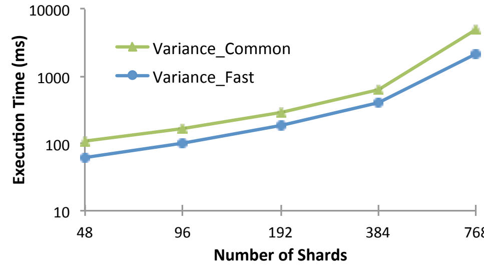


Fig. 10: Comparison of $Variance_{Fast}$ and $Variance_{Common}$.

5.3.7. Hotspots – Where is Time Spent in Sampled Aggregations?. We investigate the distribution of query execution time amongst the different phases of sampled aggregations (Figure 11). We first enumerate the various steps where a significant amount of time is spent. Before actually running the queries, we formulate a plan for doing so, taking into consideration the already cached results (time denoted by $Time_{PreQuery}$). Next, we run the queries on the shards if needed ($Time_{Query}$) and finally aggregate the results from individual shards ($Time_{PostQuery}$). We can see that $Time_{PostQuery}$ is an expensive step. We can also note that $Time_{PreQuery}$ is negligible.

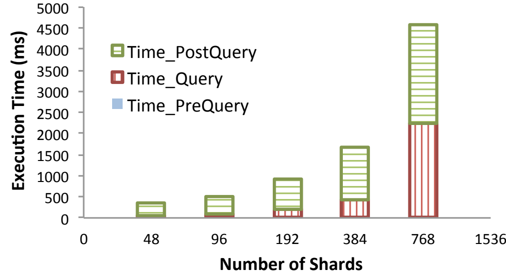


Fig. 11: Distribution of Execution Time in Sampled Aggregations.

5.3.8. Naive Aggregation vs Aggregation Tree. As mentioned above, once the queries are run on individual shards, their results need to be combined to present a single result to the user. In doing so, an obvious solution is to union the results, followed by a regrouping $(ALGO_{NAIVE_AGGREGATION})$. Another option is to combine the results using an aggregation tree $(ALGO_{TREE_AGGREGATION})$. Using an aggregation tree helps divide the workload over multiple cores, while introducing a co-ordination expense between the threads, as well as the need to write temporary results to memory. Even in the presence of the additional expenses, Figure 12 shows that $ALGO_{TREE_AGGREGATION}$ synergizes well with *Sesame*, and needs approximately half the time compared with $ALGO_{NAIVE_AGGREGATION}$.

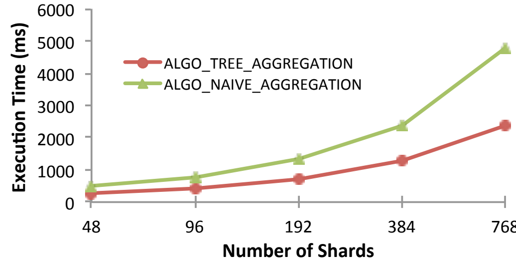


Fig. 12: Naive Aggregation vs Aggregation Tree.

5.3.9. Impact of Cores. We performed a scale-up experiment where the number of cores were increased, while other factors were kept the same (Figure 13). We can see that the execution time decreases with an increase in the number of cores. We note that the time taken for non-parallelizable code is trivial compared with the time taken to actually run the queries. The reason for non-linear increase in speedup with increasing number of cores is that the only scale-up is with regards to the number of cores – the other hardware aspects stay the same. Hence, with a simple linear increase in the number of cores, the speedup cannot be expected to be linear.

This experiment inspires us towards an interesting observation. Using 48 cores instead of 6 provides a speedup of nearly 2. On the other hand, our speculation-based techniques provide a speedup of up to 25. In this fashion, speculation can be considered as a cost-saving measure to achieve lower execution times in some circumstances (load on the server is low).

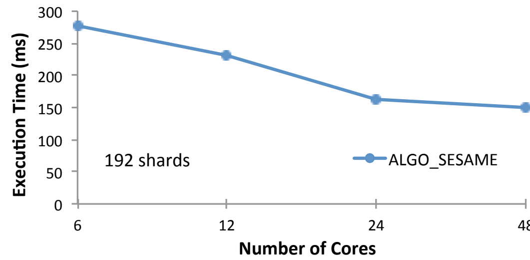


Fig. 13: Effect of Increasing Number of Cores ($Workload_{Real}$).

5.3.10. Impact of Multi-user Workloads. We have so far demonstrated the value of *Sesame* from a single user's perspective. However, the speculation techniques mentioned in this paper are applicable to multi-user contexts as well (Figure 14). Processing queries from multiple users simultaneously presents opportunities for cross-user cache reuse for user queries as well as speculative queries. To study this aspect, we used 5 query sessions based on the exploratory queries of the graduate students described earlier, each consisting of 10 queries. In the first experiment (termed as *separately*), the query sessions were run independently one after the other, with a separate cache for each session. In the second experiment (termed as *simultaneously*), queries from all sessions were run simultaneously using the same cache. Once a query finished execution, the next query in the session was scheduled.

We can see that speculative queries took around 20% lesser time. For user queries, we obtained a latency improvement of 19% at 384 shards, while there was no significant difference in performance for smaller data sizes. The reason for improved performance for 384 shards is that not all results at this sampling rate could be retrieved from the cache in the case of the queries being run separately. However, when the queries and speculative queries for multiple users are run as part of the same session, it is possible to leverage the common speculative queries across multiple users and run more speculative queries as a result. This results in a larger fraction of the user queries being answered from the cache.

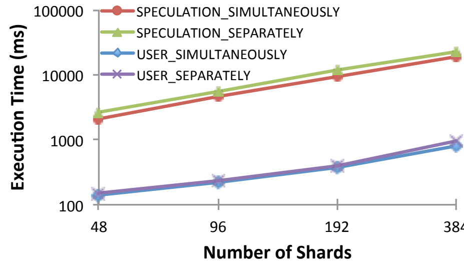
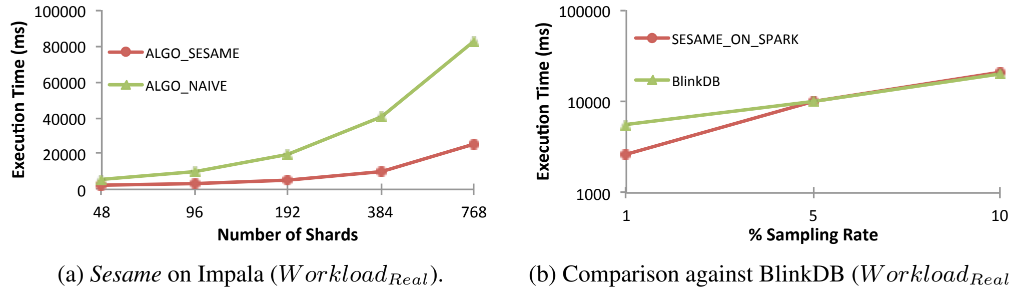


Fig. 14: Effect of Multiple Users.

5.3.11. Distributed Execution Backend: Impala. One benefit of having a rewrite-based infrastructure is the ability to study its performance in a distributed environment. Figure 15a shows the performance of *Sesame* with Cloudera Impala 1.2.4 (latest available version on Amazon EC2 EMR) as the backend, using 1 master and 15 slave nodes on Amazon EC2 of type `i2.2xlarge`, each having 8 cores (2.5 GHz), 61 GB of memory, and 2×800 GB of SSD with the maximum speculation duration set at 90 seconds. We can see that *Sesame* scales well in a distributed environment, giving speedups of up to 4.03. While these benefits are clearly significant, they are not as high as those using a

single-node PostgreSQL server – having a distributed backend impacts *Sesame*'s caching layer as well, which introduces additional latency and overhead. Architectural optimizations for distributed contexts would be ideal future work.



(a) *Sesame* on Impala ($Workload_{Real}$). (b) Comparison against BlinkDB ($Workload_{Real}$).

Fig. 15: *Sesame* on Distributed Backends.

5.3.12. Distributed Execution Backend: Spark. We also ran *Sesame* on Spark to compare our system's performance with that of BlinkDB (available at `http://blinkdb.org`), as given by Figure 15b. Both *Sesame* and BlinkDB were run on Spark 1.1.1 in the distributed environment described earlier, with the maximum speculation duration set at 90 seconds. We can see that both systems perform comparably with increasing data size. It should be noted that while BlinkDB estimates variance using the *error profile* of a smaller subset, *Sesame* computes the *exact variance* over the entire sample (detailed comparison provided in Section 1). Additionally, *Sesame*'s performance can be improved further by tuning it specifically for Spark, as opposed to our current backend-agnostic approach.

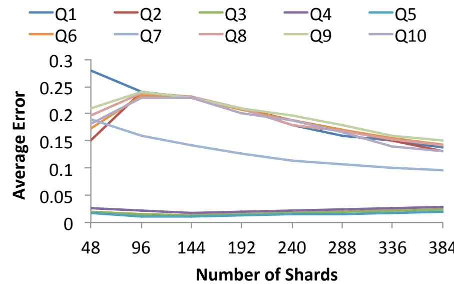


Fig. 16: Errors in Queries of a Session.

5.3.13. Average Errors in a Sample Session. To get an idea about the result quality, we looked at the average errors for one of the real-world query sessions, which consisted of 10 queries (Section 5.1). We can see that with increasing sampling rates, the error decreases in most cases. When the average error is extremely low, the increase in sampling rate might cause the average error to increase by a small amount. For some other queries, the average error increases when the sampling rate increases from 48 shards to 96 shards. These occurrences are mostly due to the introduction of new groups with slightly higher errors. Since the error rates of individual queries are distinct, and also have different patterns with increasing sampling rates, it is difficult to quantify the benefit of increased sampling rates beforehand (which is the basis of our Accuracy Gain heuristic). We note that the average session error for 384 shards, i.e. sampling rate of 50%, was around 10%.

5.4. Insights

We have seen the benefits of reusing variance computation to speed up query execution using speculative querying, and studied the benefits offered by each of *Sesame*'s features. A noticeable insight is the high penalty paid in execution time initially when the *Cache Hit Rate* falls below 100%. This is due to the fact that the overall response time is governed by the time to execute the query on all shards. Having an initial drop in *Cache Hit Rate* results in higher response times for a few queries, bumping up the overall response time. We believe we can mitigate these effects by improving our query scheduling mechanism by scheduling queries that need to access underlying data before those that can reuse the results.

The variation in query response time of *Sesame* is quite low. This can be evidenced by the fact that the coefficient of variation is at most 3% for $ALGO_{NAIVE}$, $ALGO_{SESAME}$, and $ALGO_{FULL_SPEC}$ for both datasets. For this reason, the error bars are not clearly visible, though they are displayed in most of the cases.

5.5. User Behavior

The efficacy of our faceted query model, which has been designed using the data cube traversals of roll-up, drill-down, and pivot, has been verified using two real-world query logs. In *DICE*, we analyzed a query log of ad-hoc analytical queries issued by real users, which showed that 22.97% of the queries formed sessions. All subsequent queries were covered by our faceted model, with majority of the traversals being sibling traversals. Our analysis of users using Tableau to explore the TPC-DS dataset gave us an interesting breakdown of traversal patterns. All subsequent queries were based on our traversal model, with the breakdown of individual types being as follows: Parent (26.67%), Sibling (17.78%), Child (55.56%), and Pivot (0%). We attribute the non-usage of Pivot to the users not being comfortable with this comparatively complex traversal.

5.6. Possible Impact on a User

The primary objective of *Sesame* is to accelerate execution of sampled aggregation queries by acting as a middleware between the frontend and the backend. In our experiments over data consisting of 384 shards (10.35 GB, 38400000 tuples), we were able to reduce the execution time from 7109 ms for $ALGO_{NAIVE}$ to 401 ms for $ALGO_{SESAME}$ ($Workload_{TPCDS}$) – improvements in user activity and dataset coverage as a result of providing results within 500 ms have been well-documented [Liu and Heer 2014]. For 768 shards, while latencies are no longer within interactive response times, they were still reduced by around $6.3\times$, from 14120 ms to 2212 ms. As the dataset size increases further, speculation will provide reduced speedups – here, user guidance based on the speculated results looks to be an ideal avenue for future work. We can draw similar conclusions for $Workload_{Real}$ as well, due to its similar performance.

6. RELATED WORK

Data Cubing: The concept of the data cube, operators to explore it, and the techniques to construct it were first proposed in the seminal work by Gray et al. [Gray et al. 1997], and has since found wide use in diverse contexts, including peer-to-peer systems [Kalnis et al. 2002], rule mining [Kamber et al. 1997], outlier detection [Knorr and Ng 1997; Zaiane et al. 1998], etc. In *Sesame*, instead of computing the expensive data cube in an offline step, we construct parts of it online in anticipation of relevant user queries. Determining which views and indexes to construct, given constraints such as space and maintenance costs, has been studied before [Agrawal et al. 2000; Harinarayan et al. 1996; Ross et al. 1996]. In contrast, we select the views to be materialized based on our prediction of the next user query. Materializing samples [Li et al. 2008] and succinct representations [Sismanis et al. 2002] of the cube have also been investigated. In a similar vein as offline sample construction, a partially constructed offline cube will also speed up *Sesame*'s execution engine. Tableau, an industrial software based off of Polaris [Stolte et al. 2002], uses their language for database visualization, VizQL, for exploring large multi-dimensional data cubes, by constructing an in-memory

data cube that helps rapid querying and provides quick response to changing query filters. In contrast, *Sesame* avoids the expensive offline construction of data cubes. Smart Drill-Down [Joglekar et al. 2015] presents a novel cube operator, which using a list of rules, describes interesting parts of a cube and helps the user interactively explore relational tables. Interestingness-based concepts are also presented in i^3 [Sarawagi and Sathe 2000] and can be incorporated in *Sesame* by including an interestingness-based exploration layer, which can be accelerated by our prefetching and execution engine.

Data Sampling and Approximate Querying: Generating samples under constraints such as space [Agarwal et al. 2013; Chaudhuri et al. 2007; Gibbons et al. 1998], bounded-size sampling schemes [Gemulla et al. 2013], statistical error [Sidirourgos et al. 2011], query response time [Olston et al. 2009], etc. has been looked at previously. *Sesame* focuses on online sampling and can incorporate such offline sampling techniques to improve the overall quality of the samples. While we consider queries containing `SELECT`, `PROJECT`, and `SAMPLE` operators, ideas presented in [Nirkhiwale et al. 2013; Xu et al. 2008] can be used to extend *Sesame* for relational joins.

View Reuse: There has been significant work in query rewriting using materialized views [Chirkova and Yang 2011; Halevy 2001; Kotidis and Roussopoulos 1999; Mami and Bellahsene 2012; Perez and Jermaine 2014] for general purpose, warehousing, and interactive contexts; *Sesame* draws from this body of work to design our session and sampling-aware caching strategies. We articulate the various choices for view selection in an online setup in a session-aware manner, and provide suggestions that help select the fastest option.

Prefetching: Prefetching has long been used to provide query speedups at various levels of the computational stack such as caching [Annavaram et al. 2001; Chen et al. 2007], analytical queries [Dimitriadou et al. 2014], social network [Wang et al. 2015b], spatial data [Tauheed et al. 2012], itemset mining [Baralis et al. 2013], etc. Caching and prefetching query results based on a user's query patterns has also been considered before [Battle et al. 2013; Kamat et al. 2014; Smith 1978; Zhang et al. 2001]. However, in the context of sampled aggregation, the computation of error has always been a bottleneck, which our work addresses.

Online Aggregation & Progressive Sampling: The initial efforts in online aggregation [Hellerstein et al. 1999, 1997] have been extended to continuous, parallel, and distributed contexts [Barnett et al. 2013; Condie et al. 2010; Pansare et al. 2011; Qin and Rusu 2013; Wang et al. 2015a; Wu et al. 2009]. Progressive sampling has been used to determine the optimal sample size through dynamic increments [Elomaa and Kääriäinen 2002; Estrada and Morales 2004; Gu et al. 2001; John and Langley 1996; Provost et al. 1999]. These ideas can be used to inform *Sesame* users of ideal sampling rates.

7. LIMITATIONS

Materialization of complete offline data cubes is expensive from both computation time and storage space perspectives. Cubes can also be indexed, resulting in user queries being answered more quickly, while further increasing both time and space costs. On the other hand, while *Sesame* does not need offline resources, it does need them during the query session. It is also dependent on the user following a logical exploration path in his query session, as opposed to random queries, which indexed data cubes can help with. Further, in order to reuse error in addition to the measure, variance of the measure needs to be computable in a closed form, which limits the possible reusable measures, as explained in Section 4.3.

Another drawback of a speculative execution framework is that the number of different speculative queries can increase linearly with respect to the number of dimensions. As a result, datasets having a higher number of dimensions will result in reduced speculation effectiveness. However, our modality of neighborhood-based speculation is unaffected by the dimension count being high – it is dependent on the hierarchy count, as we speculate only over nearby regions of the hierarchy. Note that a hierarchy can consist of several dimensions. For example, in our illustrative schema provided in Section 2, there exist six non-measure dimensions (*zone*, *datacenter*, *rack*, *month*, *week*, and *hour*)

and two hierarchies (*location* and *time*). To combat increasing number of hierarchies, it is possible to use smarter caching strategies based on the interestingness of the speculative queries [Joglekar et al. 2015; Sarawagi and Sathe 2000].

8. CONCLUSION & FUTURE WORK

We have presented *Sesame*, a novel multi-threaded framework that leverages holistic optimization of the query session to reduce the response time by up to an order of magnitude, when performing aggregations over sampled data. *Sesame* is the first work to look into the algebraic properties of *variance* as part of its cache reuse mechanism. We show how using different representations of variance affects query response time, and suggest using the representation that provides the fastest results in the context of interactive query execution. We note that speculative query execution depends primarily on two factors – speculation accuracy and speculation duration. Speculation duration is dependent on the time taken by the user to peruse the results, and is beyond our purview. Speculation efficiency is dependent on the user behavior and the query model, and can be improved using query logs that reflect the user's behavior better.

A complementary next step would be the investigation of faster techniques for post-aggregation of results from the different shards, since this is an important and expensive step in sampled aggregation queries. We plan to look into database backends that support hierarchical aggregation trees, multi-query optimization, and query pipelining approaches for this.

Another major avenue for future work is user guidance. Speculative query execution results in parts of the cube near the current user query being computed. This can be considered as a partial cube materialization strategy. Useful information can be provided to the user by mining the partially materialized cube to discover facts such as interesting groups [Sarawagi and Sathe 2000]. Further, finding that a region has a higher interestingness quotient helps us devise strategies such as speculatively executing queries belonging to it at a higher sampling rate, which can also help guidance and query steering systems such as *AIDE* [Cetintemel et al. 2013; Dimitriadou et al. 2014] and *SCOUT* [Tauheed et al. 2012].

Any pre-defined operator that is independent of the data and is based on the relational algebra model can be plugged into *Sesame*, since speculative execution is expedited when the operator has a superset operator. However, even for a data dependent operator such as `OUTLIER`, which can be defined in terms of the standard deviation, the reuse framework of *Sesame* can be used as long as the underlying computations are reusable. This presents an ideal opportunity for future work.

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation, under grants 1453582 and 1422977.

REFERENCES

Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: Queries With Bounded Errors and Bounded Response Times on Very Large Data. *EuroSys* (2013).

Sanjay Agrawal, Surajit Chaudhuri, and Vivek R Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. *VLDB* (2000).

Murali Annavaram, Jignesh M Patel, and Edward S Davidson. 2001. Data Prefetching by Dependence Graph Precomputation. *ISCA* (2001).

Lee Averell and Andrew Heathcote. 2011. The Form of the Forgetting Curve and the Fate of Memories. *Mathematical Psychology* (2011).

Brian Babcock, Surajit Chaudhuri, and Gautam Das. 2003. Dynamic Sample Selection for Approximate Query Processing. *SIGMOD* (2003).

Elena Baralis, Tania Cerquitelli, Silvia Chiusano, and Anais Grand. 2013. P-Mine: Parallel Itemset Mining on Large Datasets. *ICDEW* (2013).

Mike Barnett, Badrish Chandramouli, Robert DeLine, Steven Drucker, Danyel Fisher, Jonathan Goldstein, Patrick Morrison, and John Platt. 2013. Stat!-An Interactive Analytics Environment for Big Data. *SIGMOD* (2013).

Leilani Battle, Michael Stonebraker, and Remco Chang. 2013. Dynamic Reduction of Query Result Sets for Interactive Visualization. *Big Data, 2013 IEEE International Conference on* (2013).

Ugur Cetintemel, Mitch Cherniack, Justin DeBrabant, Yanlei Diao, Kyriaki Dimitriadou, Alexander Kalinin, Olga Papaemmanouil, and Stanley B Zdonik. 2013. Query Steering for Interactive Data Exploration. *CIDR* (2013).

Tony F Chan, Gene H Golub, and Randall J LeVeque. 1982. Updating Formulae and a Pairwise Algorithm for Computing Sample Variances. *COMPSTAT* (1982).

Badrish Chandramouli, Jun Yang, and Amin Vahdat. 2006. Distributed Network Querying with Bounded Approximate Caching. *DASFAA* (2006).

Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. 2007. Optimized Stratified Sampling for Approximate Query Processing. *TODS* (2007).

Shimin Chen, Anastassia Ailamaki, Phillip B Gibbons, and Todd C Mowry. 2007. Improving Hash Join Performance through Prefetching. *TODS* (2007).

Rada Chirkova and Jun Yang. 2011. Materialized Views. *Foundations and Trends in Databases* (2011).

William Cochran. 2007. *Sampling Techniques*. Wiley & Sons.

Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, John Gerth, Justin Talbot, Khaled Elmeleegy, and Russell Sears. 2010. Online Aggregation and Continuous Query Support in Mapreduce. *SIGMOD* (2010).

Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. 2014. Explore-by-Example: An Automatic Query Steering Framework for Interactive Data Exploration. *SIGMOD* (2014).

Tapio Elomaa and Matti Kääriäinen. 2002. Progressive Rademacher Sampling. *AAAI/IAAI* (2002).

Alfonso Estrada and Eduardo F Morales. 2004. NSC: A New Progressive Sampling Algorithm. *workshop on Machine Learning for Scienific data Analysis (IBERAMIA 2004)* (2004).

Minos N Garofalakis and others. 2001. Approximate Query Processing: Taming the TeraBytes. *VLDB* (2001).

Rainer Gemulla, Peter J Haas, and Wolfgang Lehner. 2013. Non-Uniformity Issues and Workarounds in Bounded-Size Sampling. *VLDB* (2013).

Phillip B Gibbons, Viswanath Poosala, Swarup Acharya, Yair Bartal, Yossi Matias, S Muthukrishnan, Sridhar Ramaswamy, and Torsten Suel. 1998. AQUA: System and Techniques for Approximate Query Answering. *Bell Labs TR* (1998).

Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery* (1997).

Baohua Gu, Bing Liu, Feifang Hu, and Huan Liu. 2001. Efficiently Determining the Starting Sample Size for Progressive Sampling. *ECML* (2001).

Alon Halevy. 2001. Answering Queries Using Views. *VLDB* (2001).

Nicolas Hanusse, Sofian Maabout, and Radu Tofan. 2011. Revisiting the Partial Data Cube Materialization. *Advances in Databases and Information Systems* (2011).

Venky Harinarayan, Anand Rajaraman, and J. D. Ullman. 1996. Implementing Data Cubes Efficiently. *SIGMOD* (1996).

Joseph M Hellerstein, Ron Avnur, Andy Chou, Christian Hidber, Chris Olston, Vijayshankar Raman, Tali Roth, and Peter J Haas. 1999. Interactive Data Analysis: The Control Project. *Computer* (1999).

Joseph M Hellerstein, Peter J Haas, and Helen J Wang. 1997. Online Aggregation. *SIGMOD* (1997).

Manas Joglekar, Hector Garcia-Molina, and Aditya Parameswaran. 2015. Smart Drill-Down: A New Data Exploration Operator. *VLDB* (2015).

George H John and Pat Langley. 1996. Static Versus Dynamic Sampling for Data Mining. *KDD*

(1996).

Panos Kalnis, Wee Siong Ng, Beng Chin Ooi, Dimitris Papadias, and Kian-Lee Tan. 2002. An Adaptive Peer-To-Peer Network for Distributed Caching of OLAP Results. *SIGMOD* (2002).

Niranjan Kamat, Prasanth Jayachandran, Karthik Tunga, and Arnab Nandi. 2014. Distributed and Interactive Cube Exploration. *ICDE* (2014).

Micheline Kamber, Jiawei Han, and Jenny Chiang. 1997. Metarule-Guided Mining of Multi-Dimensional Association Rules Using Data Cubes. *KDD* (1997).

Anja Klein, Rainer Gemulla, Philipp Rösch, and Wolfgang Lehner. 2006. Derby/s: A DBMS for Sample-Based Query Answering. *SIGMOD* (2006).

Edwin M Knorr and Raymond T Ng. 1997. A Unified Notion of Outliers: Properties and Computation. *KDD* (1997).

Donald E Knuth. 2014. *Art of Computer Programming, Volume 2: Seminumerical Algorithms, The*.

Marcel Kornacker and Justin Erickson. 2012. Cloudera Impala: Real-Time Queries in Apache Hadoop, For Real. (2012).

Yannis Kotidis and Nick Roussopoulos. 1999. DynaMat: A Dynamic View Management System for Data Warehouses. *SIGMOD* (1999).

Xiaolei Li, Jiawei Han, Zhijun Yin, Jae-Gil Lee, and Yizhou Sun. 2008. Sampling Cube: A Framework for Statistical OLAP over Sampling Data. *SIGMOD* (2008).

Zhicheng Liu and Jeffrey Heer. 2014. The Effects of Interactive Latency on Exploratory Visual Analysis. *TVCG* (2014).

Steve Lohr. 2012. The Age of Big Data. *New York Times* 11 (2012).

Elzbieta Malinowski and Esteban Zimányi. 2008. *Advanced Data Warehouse Design: From Conventional to Spatial and Temporal Applications*. Springer.

Imene Mami and Zohra Bellahsene. 2012. A Survey of View Selection Methods. *SIGMOD* (2012).

James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela H Byers. 2011. Big Data: The Next Frontier for Innovation, Competition, and Productivity. (2011).

Arnab Nandi, Cong Yu, Philip Bohannon, and Raghu Ramakrishnan. 2012. Data Cube Materialization and Mining over MapReduce. *TKDE* (2012).

Raymond T Ng, Alan Wagner, and Yu Yin. 2001. Iceberg-Cube Computation with PC Clusters. *SIGMOD* (2001).

Supriya Nirkhiwale, Alin Dobra, and Christopher Jermaine. 2013. A Sampling Algebra for Aggregate Estimation. *VLDB* (2013).

Christopher Olston, Edward Bortnikov, Khaled Elmeleegy, Flavio Junqueira, and Benjamin Reed. 2009. Interactive Analysis of Web-Scale Data. *CIDR* (2009).

Niketan Pansare, Vinayak R Borkar, Chris Jermaine, and Tyson Condie. 2011. Online Aggregation for Large Mapreduce Jobs. *PVLDB* (2011).

Luis L Perez and Christopher M Jermaine. 2014. History-Aware Query Optimization with Materialized Intermediate Views. *ICDE* (2014).

Meikel Poess, Raghunath Othayoth Nambiar, and David Walrath. 2007. Why You Should Run TPC-DS: A Workload Analysis. *VLDB* (2007).

Foster Provost, David Jensen, and Tim Oates. 1999. Efficient Progressive Sampling. *SIGKDD* (1999).

Chengjie Qin and Florin Rusu. 2013. Parallel Online Aggregation in Action. *SSDBM* (2013).

Philipp Rösch and others. 2013. Optimizing Sample Design for Approximate Query Processing. *IJKBO* (2013).

Kenneth A Ross, Divesh Srivastava, and S Sudarshan. 1996. Materialized View Maintenance and Integrity Constraint checking: Trading Space for Time. *SIGMOD* (1996).

Prasan Roy, Srinivasan Seshadri, S Sudarshan, and Siddhesh Bhobe. 2000. Efficient and Extensible Algorithms for Multi Query Optimization. *SIGMOD* (2000).

Carsten Sapia. 1999. On Modeling and Predicting Query Behavior in OLAP Systems. *DMDW* (1999).

Sunita Sarawagi and Gayatri Sathe. 2000. i3: Intelligent, Interactive Investigation of OLAP Data Cubes. *SIGMOD* (2000).
B. Shneiderman. 1984. Response Time and Display Rate in Human Performance with Computers. *CSUR* (1984).
Lefteris Sidirourgos, Martin L Kersten, and Peter A Boncz. 2011. SciBORQ: Scientific Data Management with Bounds On Runtime and Quality. *CIDR* (2011).
Yannis Sismanis, Antonios Deligiannakis, Nick Roussopoulos, and Yannis Kotidis. 2002. Dwarf: Shrinking the Petacube. *SIGMOD* (2002).
A.J. Smith. 1978. Sequentiality and Prefetching. *TODS* (1978).
Chris Stolte, Diane Tang, and Pat Hanrahan. 2002. Polaris: A System for Query, Analysis, and Visualization of Multidimensional Relational Databases. *TVCG* (2002).
Farhan Tauheed, Thomas Heinis, Felix Schürmann, Henry Markram, and Anastasia Ailamaki. 2012. SCOUT: Prefetching for Latent Structure Following Queries. *VLDB* (2012).
D. Tunkelang. 2009. Faceted Search. *Synthesis Lectures on Information Concepts, Retrieval, and Services* (2009).
Yi Wang, Linchuan Chen, and Gagan Agrawal. 2015a. Supporting Online Analytics with User-Defined Estimation and Early Termination in a MapReduce-like Framework. *Proceedings of the 2015 International Workshop on Data-Intensive Scalable Computing Systems* (2015).
Yichuan Wang, Xin Liu, David Chu, and Yunxin Liu. 2015b. EarlyBird: Mobile Prefetching of Social Network Feeds via Content Preference Mining and Usage Pattern Analysis. *Proceedings of the 16th ACM International Symposium on Mobile Ad Hoc Networking and Computing* (2015).
BP Welford. 1962. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics* (1962).
Sai Wu, Shouxu Jiang, Beng Chin Ooi, and Kian-Lee Tan. 2009. Distributed Online Aggregations. *VLDB* (2009).
Fei Xu, Christopher Jermaine, and Alin Dobra. 2008. Confidence Bounds for Sampling-based Group by Estimates. *TODS* (2008).
Osmar R Zaiane, Man Xin, and Jiawei Han. 1998. Discovering Web Access Patterns and Trends by Applying OLAP and Data Mining Technology on Web Logs. *Research and Technology Advances in Digital Libraries, 1998. ADL 98. Proceedings. IEEE International Forum on* (1998).
Kai Zeng, Shi Gao, Jiaqi Gu, Barzan Mozafari, and Carlo Zaniolo. 2014. ABS: A System for Scalable Approximate Queries with Accuracy Guarantees. *SIGMOD* (2014).
Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. 2001. On Supporting Containment Queries in Relational Database Management Systems. *SIGMOD* (2001).

APPENDIX

A. Querying over Table Shards: The interactive nature of our use case necessitates approximation of results by executing queries over a subset of the data – we use *sharded tables* for data subsetting. A sharded table contains a subset of the rows of a SQL table – concatenation of all shards is equivalent to the table. A sharded table is the atomic unit of data in our system: updates are performed at the granularity of shards, and each session makes the assumption that the list of shards and the shards themselves do not change. A sample of the data is constructed online by choosing random table shards during run-time, allowing for random sampling. We now briefly discuss how our sharding techniques maintain randomness in the samples, which is essential for obtaining meaningful error estimates.

Random Samples and Sharding: A random sample needs to satisfy two criteria. The first criterion states that every tuple should have the same probability of occurrence in the sample. The second criterion states that all possible samples of a given size should be equally likely, and thus, inclusion of a tuple in the sample should not affect the probability of inclusion of another [Cochran 2007]. *Sesame* randomizes the tuple sequence before splitting the data horizontally into shards, thereby satisfying the first criterion. As tuple indexes are independent of one another due to randomization,

the second criterion is satisfied as well. This results in every shard being a random sample. Further, as the union of random samples is a random sample, union of shards results in a random sample as well. Note that the principled approach towards sampling involves computing separate samples for each query, which is expensive and can require a full table scan. Thereby, to avoid correlation due to groups of tuples or samples being used repeatedly, a common technique is to randomize the underlying tuple sequence and create new samples periodically [Agarwal et al. 2013], which *Sesame* employs.

B. Additional Time for Calculating Variance: The queries in $Workload_{Real}$ were run on 10 shards using the in-memory PostgreSQL setup described in Section 5. Average time taken when using `SUM`, `AVERAGE`, and `COUNT` was 989.5 ms, whereas it was 1529.2 ms when including `VARIANCE` as well. Thus, inclusion of variance resulted in the computation requiring 54.6% more time.