Distributed and Interactive Cube Exploration

Niranjan Kamat [#1], Prasanth Jayachandran [#2], Karthik Tunga [#3], Arnab Nandi [#4]

[#] *Computer Science and Engineering Department, The Ohio State University*
2015 Neil Avenue, Columbus, OH 43210, USA
{kamatn[1], jayachan[2], tunga[3], arnab[4]}@cse.osu.edu

***Abstract*—Interactive ad-hoc analytics over large datasets has become an increasingly popular use case. We detail the challenges encountered when building a distributed system that allows the interactive exploration of a data cube. We introduce *DICE*, a distributed system that uses a novel session-oriented model for data cube exploration, designed to provide the user with interactive sub-second latencies for specified accuracy levels. A novel framework is provided that combines three concepts: faceted exploration of data cubes, speculative execution of queries and query execution over subsets of data. We discuss design considerations, implementation details and optimizations of our system. Experiments demonstrate that *DICE* provides a sub-second interactive cube exploration experience at the billion-tuple scale that is at least 33% faster than current approaches.**

I. Introduction

Large-scale analytics has found a growing number of use cases in a variety of disciplines, from business to the sciences. With the rapid rise in data, and the reliance on data-driven insights for decision making, planning and analysis, the role of analytics over massive datasets has become a critical one.

With the proliferation of large-scale data infrastructure, it is not uncommon for *end-users* to expect direct fine-grained control over large amounts of data. The availability of both dedicated and dynamically provisioned distributed computational resources allows analyses that were typically handled by database administrators to be performed by the end-users of the analyses themselves. Further, there is an increasing demand in real-time or near-real-time analytics, where all analysis is performed on in-situ data, such as constantly-updating logs that are being appended to in a batched manner. As detailed in the following paragraphs, despite the availability of performant, distributed and scalable infrastructure, there exist several challenges to large-scale analytics.

In addition to the typical use cases of reporting, where predetermined query templates are run over batches of new incoming data, and mining, where data is analyzed to discover interesting patterns of information, there has been a sharp rise in the demand for **ad-hoc analytics**, exposed to the user over interfaces for business intelligence, interactive dashboards and advanced domain-specific data-driven applications. These challenges are exacerbated in the scope of ad-hoc analysis over a `CUBE` representation [19] of the data. Such a representation is useful for the purpose of exploratory data analysis, since successive investigatory questions can be answered in the form of *drilldown* or *rollup* queries.

Data cube exploration is often expected to be *interactive* – queries need to be responded to within a small **latency bound**. Studies in human-computer interaction [36], [47] establish guidelines and demonstrate the functional and economic value of rapid response times, heavily motivating a sub-1000 ms (i.e. sub-second) threshold for the database to respond to the user. For our system, we empirically observed that latencies of up to 1000 ms were perceived as fluid, and it took around 5000 ms for the user to view and react to the query results.

A. Common Approaches

Intuitively, the simplest approach to ensuring fast, interactive cube exploration is to *materialize the entire data cube* such that each query to the cube is simply a lookup from a main memory cache. While such a setup will perform within the latency bounds we are subject to, we are constrained by scale: a fully materialized cube can be several multiples of the original dataset, which typically is larger than available memory. Further, such a strategy does not work in the case of ad-hoc (e.g. computed) dimensions or if the user is inspecting a new measure. Thus, an often-used approach is to execute the query over an offline computed sample of the data [2], [10], [32], [54]. However, this approach cannot accommodate changes in the underlying data. The techniques described in our paper are complementary to such an approach and can easily be adapted, if needed, to accommodate offline sampling. *Online aggregation* approaches have also been studied [24], but require a significant overhaul of the entire query processing infrastructure. Further related work is provided in Section V.

This paper introduces *DICE*, a system that proposes a ***session-oriented*** approach to data cube exploration that caters to the challenges observed. In contrast to existing OLAP systems, our system is designed keeping in mind the *user's flow*, surfacing approximate results within interactive latencies.

Contributions:

- We introduce *DICE*, a distributed system that allows exploration of 1-billion-tuple data cubes at **sub-second levels**.
- We provide a principled cost-based framework that combines two complementary techniques: **speculative query execution** and **online data sampling** to achieve *interactive* latencies for cube exploration in a distributed framework.
- To *bound the space of possible speculative queries*, we propose a ***faceted*** **cube exploration model** that considers successive queries as part of a *query session*.
- We share insights into the design and implementation of our system based on **real-world query logs, user studies and detailed performance evaluations**.

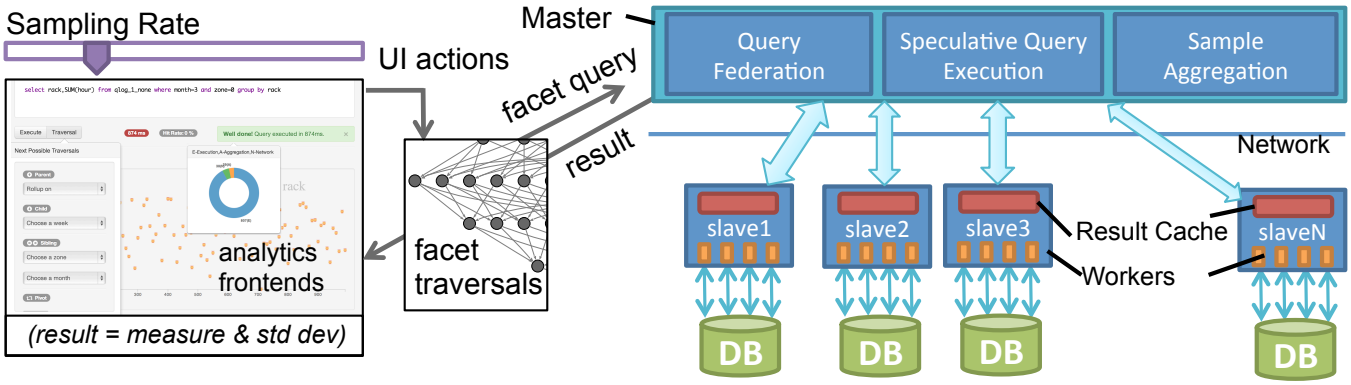

Fig. 1. **DICE Approach:** Allow *tunable sampling rates* on low-latency frontends, with UI actions translated to *facet traversal queries* (Section II-A) over the data cube. Queries are executed by the master over distributed slaves (Section II-B). In the DICE approach (Section III), the master manages session state, query speculation and result aggregation, while the slaves manage execution and caching. For each query, the master distributes the query to each slave, which may have some results speculatively executed and cached. Results from each slave are then aggregated, error bounds are calculated, and returned to the user.

B. Motivating Example

One typical use of interactive cube exploration is in the *management of cloud infrastructure*. For each setup, a handful of operations personnel manage tens of thousands of nodes, each with multiple virtual machines. Each instance produces a plethora of events, which are logged to track performance, detect failures and investigate systems issues. Each event item can be understood as a tuple with several fields, and each analytics task can be considered as a projection on a cube over the entire dataset. Event log data is copied over from all instances into a distributed data store, and is typically queried within fixed time ranges. Queries are ad-hoc, and due to the critical nature of the task, a system that allows for fast, interactive aggregations is highly desirable. Thus, an example query in our use case can be given by:

```
SELECT rack, AVG(iops)
FROM events
WHERE datacenter = "EU" AND hour = 6
GROUP BY rack;
```

Such a query can be used to identify problematic I/O rates across racks which could cause failures in a datacenter over time. We expect such queries to be either written by the operations personnel directly, or be generated automatically by applications that provide visualizations and an easy-to-use querying layer. An important insight is that such a process is not about aiding exploration such that user intervention is not required, but about helping the user analyze data faster by reducing the time it takes to interact with the data.

Our use case is driven primarily by the need for *interactive* data cube exploration. First, querying is *ad-hoc and exploratory*. Given the variety of possible questions to be answered, it is difficult to implement such a system over traditional reporting platforms, streaming queries, incrementally materialized views or query templates. Second, the data is *distributed* due to its size and nature of generation: events from each node in the datacenter are copied over to a set of nodes dedicated to this ad-hoc analysis to be used by one or few people. Another consequence of the size of the data is that it is *impractical to construct a fully materialized cube* to perform analysis. Third, user interaction, either through the application interface or through direct querying should not impede the user in performing their exploration task. Thus, the interaction needs to be fluid, requiring the underlying queries to return quickly, enforcing the latency bounds discussed above. Given the sampling rate specified by the user, it is desirable that the results for the specified number of queries be returned at the earliest. Lastly, queries are seldom one-off, and almost always occur as part of a larger *session of related queries*. In light of this characterization, our problem thus becomes: *Given a relation that is stored across multiple nodes, and the queries issued by the user so far, ensure that each query in the session is responded to at the earliest, at the user specified sampling rate.* We will formally define this problem in Section II, along with the overall data model.

II. Data Model and Preliminaries

Having motivated the problem setting of a *distributed*, *interactive*, *cube exploration* system, we now discuss preliminaries for each of these three contexts. We begin with cube exploration, where we define a *faceted exploration* model to facilitate complete yet efficient exploration of the data cube. As we will discuss in the following section, faceted exploration bounds the space of successive queries, thereby making speculative query execution feasible. Second, we discuss the execution of faceted queries in a distributed setting, where data is distributed across nodes as *table shards*. Finally, given the constraints of interactivity, we explain our techniques for approximate querying over sampled data, provide a framework to execute faceted queries over multiple nodes, and draw from concepts of stratified sampling and post-stratification to aggregate results and estimate error bounds.

A. Faceted Exploration of Data Cubes

In the context of cube exploration, the definitions of *cube*, *region*, and *group* are as per the original data cube paper [19].

A region denotes a node in the cube lattice and a group denotes tuples with the same values of attributes for that region. For example, one of the groups in the region {datacenter,month} is {EU,January} for the cube derived from the motivating example. We continue with our motivating example, using the following schema: Database table *events* catalogs all the system events across the cluster and has three dimensions, two of which are hierarchical:

location[zone:datacenter:rack], time[month:week:hour], iops

Challenges in Exploration: As a user exploring a data cube, the number of possible parts of the cube to explore (i.e. cube groups) is very large, and thus, exploration can be unwieldy. To this end, we introduce the *faceted* model of cube exploration, which simplifies cube exploration into a set of facet traversals, as described below. As we will see in the following section, the faceted model drastically reduces the space of possible cube exploration and simplifies speculative query execution, which is essential to the DICE architecture.

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 [53]. Empirically, most visualizations such as map views and bar charts found in visual analytics tools can be constructed from aggregations along a single dimension. 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.

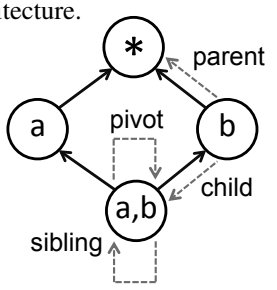

Fig. 2. Faceted Cube Exploration Traversals

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 one region, along a specific dimension. The user then explores the cube by traversing from that facet to another facet. This successive facet can be a **parent** facet in the case of a rollup, a **child** facet in the case of a drilldown, a **sibling** facet in the case of change of a dimension value in the group and 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. A session comprises of multiple traversals. The formal definitions are as follows.

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 differ on exactly one dimension d_i, i.e. $\forall g_a, g_b \in f, d_i(g_a) \neq d_i(g_b) \wedge d_j(g_a) = d_j(g_b)$ where $i \neq j$ and d_i is the *grouping dimension*, and the remaining dimensions are the *bound dimensions*. In its SQL representation, a facet in a region contains a GROUP BY on the grouping dimension and a conjunction of WHERE clauses on the bound dimensions of that region. A facet can be referred to using the notation $f(d_g, \overrightarrow{d_b : v_b})$ where $d_g \cup \overrightarrow{d_b}$ denotes the dimensions in the corresponding region, d_g denotes the grouping dimension, $\overrightarrow{d_b : v_b}$ denotes a vector representing 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 zones for a specific week and month.

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. The transition from one facet to another is known as a traversal.

We now define four traversals, *Parent*, *Child*, *Sibling* and *Pivot*, inspired by similar traversals over data cube, each allowing us to move from one facet to another. We define them in terms of the destination facet, as follows.

Parent Facet: A parent facet is defined as any facet obtained by generalizing any of the bound dimensions. Thus, a facet $f_p(d_{pg}, \overrightarrow{d_{pb} : v_{pb}})$ is a parent to the facet $f_(d_g, \overrightarrow{d_b : v_b})$ if $d_{pg} = d_g$ and $\overrightarrow{d_{pb} : v_{pb}}$ represents a parent group of $\overrightarrow{d_b : v_b}$ in the cube lattice. The parent facet $f(zone, month : m_1)$ generalizes the dimension *time* from the prior example.

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

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

Pivot Facet: A pivot facet is defined as any facet obtained by switching the grouping dimension with a bound dimension. Thus, a facet $f(d_g, \overrightarrow{d_b : v_b})$ can be pivoted to the facet $f_v(d_{vg}, \overrightarrow{d_{vb} : v_{vb}})$ if $d_{vg} \in \overrightarrow{d_b} \wedge d_g \in \overrightarrow{d_{vb}}$ and $\overrightarrow{v_b}$ and $\overrightarrow{v_{vb}}$ have all but one bound dimension and value in common. The facet $f(week, zone : z_1, month : m_1)$ pivots on *zone* z_1 from the facet example, and is therefore its pivot facet.

EXPLORABILITY OF THE CUBE: It is clear that in our model, the user is able to *fully explore* the data cube, i.e. all cube groups can be explored using facets, and it is possible to reach any facet from any other facet. First, a group $g = \overrightarrow{d : v}$, can be obtained from $|\overrightarrow{d}|$ facets, $f(d_g, \overrightarrow{d_b : v_b}) : d_g \in \overrightarrow{d} \wedge \overrightarrow{d_b} = \overrightarrow{d} - d_g$. Second, any two facets in a region can be reached from another by a series of sibling and pivot traversals: sibling traversals to change bound values, and pivot traversals to switch between bound and grouped dimensions. Parent and child traversals allow us to reach the corresponding parent and child regions in the cube lattice. Thus, the four traversals enable full exploration of the cube lattice. Note that we do not require users to follow only the listed traversals – faceted traversals simply reduce the space of successive queries for speculation (Section III-A).

EFFECTIVENESS OF FACETED MODEL: The four traversals mentioned above are **both intuitive and sufficient to explore the *entire* data cube**. The parent, child and pivot traversals are inspired by rollup, drilldown and pivot operations respectively. It is always possible to add more traversal types, especially by mining a user's query history for common "patterns" of analysis, e.g. keeping the bound dimensions the same and changing the group by dimension. Such extensions are easily pluggable into our system, but not required – the four traversals described above are intuitive and powerful enough to traverse the cube. We quantify the applicability of our model on **real-world query logs and measure user satisfaction using a user study**, described in Section IV-C.

B. Distributed Execution

The interactive nature of our use case necessitates the approximation of results by executing queries over a subset of the data. We use *sharded tables* to achieve distributed and sampled execution of queries. A sharded table contains a subset of the rows of a SQL table and the concatenation of all shards across nodes is equivalent to the entire dataset. Each node may contain multiple shards. A sharded table is the atomic unit of data in our system: updates are performed at the granularity of the shard level, and each session makes the assumption that the list of shards and the shards themselves do not change.

C. Querying over Table Shards

A sample of the data is constructed online by choosing random table shards during run-time, allowing for random sampling. We use standard sampling concepts of stratified sampling [13] and post-stratification [13] for estimating the error bounds. Details on our use of sampling methods are provided in the appendix.

Given the preliminaries and definitions, in the naive case, the problem of ad-hoc cube exploration using the facet exploration model is simply that of successively executing each query received at a given sampling rate. We formulate our problem as the following:

For a facet session $\vec{F}$, where each ad-hoc facet query f_i is expected to execute at a certain sampling rate, and the expected time between the termination of one facet query and the start of the next ad-hoc facet query (i.e., the time taken to view the results of the prior query) is τ_V, return f_i as quickly as possible to the end-user, preferably within the interactive threshold τ_I.

Accuracy Gain Heuristic: In order to schedule speculative queries at different sampling rates, we need to know the reduction in sampling error at different sampling rates. However, it cannot be known before actually sampling the data. Therefore, we construct a heuristic based on the consistency property of Maximum Likelihood Estimation (MLE), $||\theta^* - \theta|| = O(\frac{1}{\sqrt{n}})$ where θ^* is the current estimate, θ is the true value and n is the current sampling rate, which informs us that the difference between our estimate and the true value will be inversely proportional to the square root of the current sampling rate. Therefore, we can estimate the future gain in accuracy based on the sampling rate. Thus, the estimated *gain* in the accuracy due to a unit sampling rate increase can be given as

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

where R_{curr} is the current sampling rate and c is the constant from the proportionality heuristic.

With more time permissible, we issue the same query on multiple tables on multiple nodes progressively giving us a smaller standard error for the estimators. Our goal then during speculative execution of the queries is to increase the likelihood that the next user query would be cached at a higher sampling rate allowing us to retrieve the results at the desired sampling rate at the earliest. We cast this to fit the DICE framework in the following section.

III. THE DICE SYSTEM

A. Speculating Queries in a Session

A crucial insight to ad-hoc querying is that **queries occur in sessions**. Thus, it is prudent to think of improving query performance holistically at the session level. A session comprises several ad-hoc queries, each of which requires low-latency responses. The result for each query is inspected by the user for a small amount of time, after which the next query is issued. We consider this as a hidden opportunity – the database is simply waiting on the user to issue the next query. In light of this, our solution is to *utilize this waiting time to speculate, execute and cache the most likely followup queries at the highest quality possible.* While the concept of speculative execution is an intuitive one, there are several challenges to implementing it over a distributed, approximate querying environment – especially in the context of data cube exploration. The challenges comprise a host of interdependent problems: What are the most likely followup queries? What is the strategy to employ to execute and cache likely queries? In a sampling approach, what is the highest sampling rate to run a speculative query at, given interactive constraints? Finally, is there a singular framework to combine these problems into a cohesive, unified system?

Given these challenges, we present the *DICE* system that solves the problem by using three complementary strategies. First, it performs **speculative query execution**, by caching results of likely followup queries, allowing for reduced latencies for ad-hoc query sessions. The enumeration of the likely followup queries is made possible by the *faceted* model of data cube exploration described in Section II. Second, *DICE* employs a novel architecture of query execution over a distributed database, executing queries piecemeal over individual table shards and then assembling them in a post-processing step. This novel architecture in turn allows for **bounded-time execution** of queries ensuring interactive latencies. Third, it employs a **cost-based model** for the prioritized execution of speculative queries such that likely queries are executed at higher sampling rates.

B. System Architecture

The architecture of our system employs a hierarchical master-slave approach, such that all queries are issued to the master, and responded to by the master. In line with the setting described in Section II-B, each slave manages multiple *table shards*. Each shard is atomic and read-only, and is implemented as a table in a commodity relational database. The catalog of shards across all slave nodes is maintained at the master. For a single exploration session, the catalog is used to ensure that the list of shards addressed is constant. The slaves maintain an in-memory LRU cache for the results. In a fast-changing database, table shards can be atomically added and deleted from the slaves, and the master's catalog can be updated, allowing for querying over rapidly changing data.

C. Query Flow

The high-level query flow of *DICE* is as follows: each ad-hoc query is rewritten and federated to the slave nodes, where it is executed. The results are returned, aggregated and presented to the user, along with the accuracy of the query. Upon success, a set of speculative queries is executed till the next user query is received, with the goal of increasing the likelihood of caching as many of the future queries as possible. When the successive ad-hoc query is issued, it is again rewritten and federated, with the hope that its results are cached at the slaves at a high sampling rate, thus reducing the latency of the overall ad-hoc query.

User Query: At startup, the master makes sure that all the slaves are running and ready to accept queries. On receiving an ad-hoc query, the query is rewritten into multiple queries, one per required random table shard and passed to each slave. Since data is horizontally distributed across all slave nodes, the query itself is identical, with the exception of `id` of the table shard addressed. On completion of an ad-hoc query (or if the results of the query were already in the cache), each slave returns the results back to the master, where the results are aggregated, and error calculation performed, and this information presented to the user.

Speculative Queries: Upon completion of the ad-hoc query, the master immediately schedules a list of speculative queries that can be issued by the user. While the space of possible queries is unbounded, we restrict our speculations using faceted exploration framework; thus allowing the list of possible queries to be enumerable. Speculated queries are then ranked (as discussed in the following subsection), and distributed amongst the slaves in a round-robin fashion. Each slave issues, in an increasing order of rank, a predefined number of concurrent queries to its database and populates the results in its cache (speculative query results are not sent to the master). Upon receiving the next user query, the slave kills all currently running speculative queries.

Successive User Query: When the next ad-hoc query arrives, it is again rewritten and federated to the slaves. If the exact query or a unified query (refer to Section III-F) is cached, the result of the ad-hoc query is materialized from the cached result. If it is not cached it is then executed on the database. The caching of speculated queries drastically impacts ad-hoc query latency and allows for a fluid and interactive data cube exploration experience.

D. Prioritizing Speculative Queries

As is clear from the query flow and the faceted model, each ad-hoc query can yield significantly large number of speculative queries. Given the bounded time available for execution, it is typically not possible to execute all the speculative queries. Thus, it is necessary to prioritize speculative query execution such that it maximizes the likelihood of results for the successive query being returned from the cache. This can in turn be done by maximizing the overall gain in accuracy, as discussed in Section II-C. The selection of the maximal subset can be modeled as a linear integer programming problem as follows:

$$\text{MAXIMIZE: } \sum_{q \in Q} Prob(q) \cdot AccuracyGain(SR) \cdot x_q$$
$$\text{SUBJECT TO: } \sum_{q \in Q} Time(q) \cdot x_q <= totalSpecTime$$
$$\text{WHERE: } x_q \in \{0, 1\}.$$

Here, $Prob(q)$ gives the probability of a query q, which should be obtained from the query logs, Q is the set of all speculative queries at all sampling rates, $AccuracyGain(SR)$ is the estimated gain in sampling accuracy which depends on the sampling rate SR of q as described in Section II-C, $Time(q)$ is the estimated running time and $totalSpecTime$ is the expected total speculative time.

Considering the input parameters, it is not possible to solve the above optimization problem in sub-second latency thus preventing us from returning results within those latencies. We expect the majority of the query execution cost to be typically due to an in-memory table scan over identically sized data if the table shards are pre-loaded in the memory. It is not possible to load the entire dataset into memory but definitely a significant fraction which in our experiments was up to 20% such that the error bars for most of the groups were small. This lets us assume unit execution time for each query over a shard. In that case, it is clear that choosing the query that yields the maximum of the product of the probability of a query and the estimated accuracy gain for the corresponding sampling rate is the best decision. Therefore, the solution to the problem of choosing of the *best* queries that yield the highest overall accuracy gain turns into a greedy selection problem, the algorithm to which we provide in the following section.

Greedy Approach: The greedy cost-based approach prioritizes the execution of *the most likely queries that provide the highest overall accuracy gains*. We represent the *score* of a query q at the sampling rate of SR as $Prob(q) \cdot AccuracyGain(SR)$.

In the case of multi-query optimizations such as *unification* (described in Section III-F), where multiple queries are grouped together into a unified query $Q = q_{1..n}$, the score can be represented as $\sum_{q \in Q} Prob(q) \cdot AccuracyGain(q)$. Queries are run greedily on the worker nodes in descending order of the score. Since worker nodes are capable of bounded-time execution and each query runs in a time lesser than the view latency threshold due to the small size of the table shard, this approach proves to be a viable strategy and successfully provides for sub-second latencies, as observed in Section IV.

In the case of sibling traversals for ordinal dimensions, a user is more likely to choose the changed bound dimension value closer to the current value. We use a heuristic that the distribution of the probability of the value that the changing dimension in the where predicate takes can be given as $P(newVal) = O(\frac{1}{||newVal - oldVal||^2})$. Let the set of speculative sibling queries and their probabilities be $SQ = \{SQ_{1...n}\}$ and $P = \{P_{1...n}\}$ respectively. We re-distribute the sum of these probabilities between queries in SQ as $P(x) = \frac{1}{c*(x-oldVal)^2}$ where c is the normalization constant given by $\sum_{y \in \Upsilon} \frac{1}{(y-oldVal)^2}$ where Υ is the domain of the changing dimension. Using query logs, user behavior can be modeled using the above distribution as the prior.

E. The DICE Algorithm

We are now able to illustrate both the overall model of the system (Algorithm 1) and the *DICE* algorithm (Algorithm 2).

```
EXPLORE(User u)
 1  //CF : Current Facet
 2  CF = null
 3  while True
 4  do
 5      Query q ← TRAVERSE(u, CF)
 6      Results r ← EXEC(q)
 7      Q_Spec ← ENUMERATE-SPEC(q)
 8      P_Spec ← DICE-PLAN-DETERMINE(Q_Spec, CF)
 9      for each node n, queries Q in P_Spec
10      do //parallel loop till next user query
11          NODE-EXEC-ALL(n, Q)
```

Algorithm 1: Core Exploration Loop

```
DICE-PLAN-DETERMINE(Q_Spec, CF)
 1  P_Spec ← GET-SPEC-PROBABILITIES(Q_Spec)
 2  PS_Spec ← SIBLING-ADJUSTMENT(P_Spec, CF)
 3  Accuracy_Gains ← GENERATE-ACCURACY-GAINS-VECTOR()
 4  Unified_Queries ← QUERY-UNIFICATION(Q_Spec)
 5  P_Unified_Queries ← {}
 6  for each UQ in Unified_Queries
 7  do
 8      P_UQ ← Σ_{Q∈UQ} PS_Spec(Q)
 9      Unified_Accuracies = Unified_Queries ×
10                           Accuracy_Gains
11  DESC-SORT(Unified_Accuracies)
12  return Unified_Accuracies
```

Algorithm 2: DICE Execution Strategy

Algorithm 1 (Core Exploration Loop) describes the overall DICE cube exploration system. A user first selects a query (Line 5) which is then executed (Line 6). The system then enumerates all the different possible speculative queries based on the cube exploration model described earlier (Line 7) and ranks them (Line 8). It then distributes the workload across all the available nodes (Lines $9--11$). Next, in Algorithm 2, we formally describe how *DICE* ranks the speculative queries at different sampling rates.

Algorithm 2 (DICE Execution Strategy) starts by first finding out the normalized probabilities P_{spec} given a set of speculative queries (Line 1), and reweighting probabilities of the sibling queries as described in Section III-D (Line 2). Next, it generates the vector of the estimated accuracy gains for all sampling rates (Line 3) and then performs unification over all the speculative queries (Line 4) as given in Section III-F. Finally, it ranks the unified queries at different sampling rates by the product of their probabilities and their corresponding sampling rate accuracy gains (Lines $5--11$) and returns the sorted queries (Line 12).

F. Optimization: Query Unification

We now detail *unification*, a technique to speed up query execution. For each traversal, the number of speculative queries given the current facet $f(d_g, \overrightarrow{d_b : v_b})$, in the worst case is:

$$NumParent = |\overrightarrow{d_b}|$$
$$NumChild = \sum_{dim \in \{Dimensions\} - \{\overrightarrow{d_b}, d_g\}} Cardinality(dim)$$
$$NumSibling = \sum_{dim \in \overrightarrow{d_b}} Cardinality(dim) - |\overrightarrow{d_b}|$$
$$NumPivot = Cardinality(d_g) * |\overrightarrow{d_b}|$$

Consequently, one can infer that the total number of speculative queries could be greater than the sum of the cardinalities of all the dimensions. Further, taking replication of queries due to usage of table shards results into consideration, the total number of speculative queries is equal to the product of the number of table shards and the number of distinct speculative queries. Hence, it is not feasible to run all the speculative queries for most real-world datasets at high sampling rates within interactive time bounds.

We can observe that the generation of speculative queries leads to several queries that differ only by the value of a single bound dimension. Unifying multiple such queries into a lesser number of queries becomes essential since **concurrently running all of them will congest the system**. We have used two techniques of minimizing the number of queries by unification. The first is to unify `WHERE` clauses on a column into a `GROUP BY` on the column, and the second is to split a dimension's domain into ranges, and issuing range-based queries. The results of these unified queries can be post-processed to extract results for the user query.

Groupby Based Unification: Multiple queries can be unified into a single query by replacing the bound dimension that takes multiple values by a `GROUP BY` on the same dimension when the cardinality of a dimension is moderately high (i.e. above a set threshold). This unification leads to the following speculative queries for the current user facet $f(d_g, \overrightarrow{d_b : v_b})$ in a cube of dimensions $\overrightarrow{d}$:

$$Parent - Set = \{\forall d_i; d_i \in \overrightarrow{d_b} : f(d_g, \overrightarrow{d_b : v_b} - d_i : v_i)\}$$
$$Sibling - Set = \{\forall d_i; d_i \in \overrightarrow{d_b} : f(d_g, d_i, \overrightarrow{d_b : v_b} - d_i : v_i)\}$$
$$Pivot - Set = \{\forall d_i; d_i \in \overrightarrow{d_b} : f(d_g, d_i, \overrightarrow{d_b : v_b} - d_i : v_i)\}$$
$$Child - Set = \{\forall d_i; d_i \in \overrightarrow{d} - \overrightarrow{d_b} : f(d_g, d_i, \overrightarrow{d_b : v_b})\}$$

The sibling and pivot queries thus generated are identical. One can also notice that a parent query $f(d_g, \overrightarrow{d_b : v_b} - d_i : v_i)$ can be answered by the corresponding sibling/pivot query $f(d_g, d_i, \overrightarrow{d_b : v_b} - d_i : v_i)$ where $d_i : v_i \in \overrightarrow{d_b : v_b}$. Thus, groupby-based unification leads to an enormous reduction in possible queries needed to be run. However, the results for the next query would need to be retrieved from the new unified query's result set, and this post-processing may be expensive. Typically, groupby unification is useful, specifically for moderately high (thresholds set empirically) cardinality dimensions. There is, clearly, a tradeoff between running a large number of non-unified queries, and a single unified query with a large result set.

Range Based Unification:

At *very* high cardinalities, the problem of the very high number of speculative queries is not resolved by the groupby-based unification since the result set is expected to be large. Unifying the queries into *ranges* was found to be extremely useful. We convert multiple speculative queries $f(d_g, \overrightarrow{d_b : v_1..v_n})$ into fewer range-based speculative queries $f(d_g, \overrightarrow{d_b : [v_1..v_{n_1}]})$, $f(d_g, \overrightarrow{d_b : [v_{n_1}..v_{n_2}]})$ .. $f(d_g, \overrightarrow{d_b : [v_{n_{k-1}}..v_{n_k}]})$. The choice between range-based and groupby-based unification depends on the column cardinality and is a tunable parameter. This parameter can be obtained empirically using the marginal distribution of the column and the prior workload.

An interesting observation with range queries is that even with careful tuning of the ranges, the cardinality of the data for each range-unified query is large enough to motivate the use of an index on range-unified columns. Thus, we only index dimensions with very high cardinalities. While this introduces variability into our cost model, the lack of a good determiner for the cost of a range-unified query and aforementioned lack of a fast solution to our linear integer programming problem compels us to invoke Occam's Razor and use a unit cost in this case and the resultant greedy algorithm for query selection.

G. Optimality of DICE

As described in Sections III-D and III-F, we cast the linear integer programming problem of maximizing the overall accuracy under the constraint of maximum allocated time into a greedy algorithm of choosing the new query at an additional unit sampling rate. Also, as mentioned earlier in Section II-C, we would not know the accuracy gain without actually running the query. Thus, approximating the gain using the estimated accuracy gain based on the MLE Consistency property is a sound assumption to make. Therefore, the DICE algorithm of choosing a new query with the highest product of probability from the workload and the estimated accuracy gain at the newer sampling rate will indeed be the optimal strategy.

IV. Experiments and Evaluation

A. Experimental Setup

DICE is implemented in Java running on Sun Java 6 VMs and uses PostgreSQL 9.1 as the database for each slave node. By default, we discard the first run of each experiment and report the average of the following three runs (runs were nearly identical for all experiments, with no outliers, also observed by the low standard deviation). We perform an exhaustive analysis of the *DICE* system over a variety of cluster configurations, workloads and algorithms for our metrics, as described below.

Cluster Configurations: $\text{CLUSTER}_{\text{SMALL}}$ is a private cluster built on commodity hardware with only *DICE* running during the experiments. The master node has 1 Quad Core 3.30GHz Intel i5 CPU, 16GB DDR3 RAM @1333MHz & 256GB SATA HDD and the 15 slave nodes each possess 1 Quad Core 2.13GHz Intel Xeon CPU, 4GB DDR2 RAM @667MHz & 720GB SATA HDD. Nodes are connected over a Gigabit Ethernet switch. Each slave contains 4 workers. $\text{CLUSTER}_{\text{CLOUD}}$ is an Amazon EC2 configuration of 1 master and 50 slaves of the `c1.xlarge` type, each with 7GB Memory and 8 Virtual cores, powering 8 workers per slave node. All nodes for both configurations run Ubuntu Linux 12.04 LTS.

Dataset: Our generated dataset conforms to the example schema provided in Section I, and comprises **1 billion rows** sharded uniformly across all nodes with a default table shard size of 1M rows. The distributions and cardinalities are:
location[*uniform*]:[zone{10}:datacenter{100}:rack{1000}],
time[*gaussian*]:[month{12}:week{52}:hour{24}] and
iops[*zipfian*]:{10000}.
Each table shard is 102MB on disk, with a data size of 81MB and index size of 21MB, yielding in a total of 1000 table shards spanning 100GB. Unless otherwise specified, we run experiments at 20% sampling rate i.e., 200 million rows are actually processed. [1]

Workloads: The user was asked to explore the dataset taking into consideration the faceted exploration model using a popular BI tool. Query logs from the tool were used to derive the workload. A workload depicts a user query session of 10 facet traversals, with the measure function `AVG`. Unless stated, 3 workloads were used for each experiment and with the aforementioned 3 runs, results into a sample size of 90 queries. The viewing latency threshold τ_V is fixed to 5000ms.

Algorithms: We compare five different algorithms: $\text{ALGO}_{\text{NOSPEC}}$ stands for "No Speculation" and represents the baseline use case, i.e. ad-hoc distributed querying without any speculation, similar in design to modern distributed query execution engines. $\text{ALGO}_{\text{RANDOM}}$ represents distributed querying using query speculation, but the queries chosen to

[1] It should be noted that due to the variability of schema, row / columnar storage layouts and hardware performance, our focus is on the number of rows processed, and not the disk representation.

be speculated are selected randomly from the set of possible facet traversals. ALGO$_{\text{UNIFORM}}$ selects speculative queries uniformly from each *type* of facet traversal. ALGO$_{\text{DICE}}$ uses the *DICE* speculative query selection technique. ALGO$_{\text{PERFECT}}$ "improves" upon *DICE* by allowing for a **perfect** prediction of the subsequent ad-hoc query – this represents the (hypothetical) best-case performance of our speculation strategy, and is included to demonstrate the overall potential of speculative caching.

Metrics: AVERAGE LATENCY is measured in milliseconds as the average latency of a query across sessions and runs. We also depict ± 1 standard deviation of latency using error bars in most of our results. AVERAGE ACCURACY is measured as the absolute percentage deviation of the sampled results from the results over the entire dataset.

B. Results

1) Impact of Data Size: We observe, in Figure 3 & 4, the impact of data size on the latency observed by each algorithm by varying the target sample size for the ad-hoc queries in our workload. ALGO$_{\text{NOSPEC}}$ scales almost linearly, and exceeds the sub-second threshold for 200M rows. Despite issuing speculative queries, ALGO$_{\text{RANDOM}}$ and ALGO$_{\text{UNIFORM}}$ perform just as poorly as ALGO$_{\text{NOSPEC}}$, validating the need for a principled approach to speculative querying that *DICE* provides. ALGO$_{\text{DICE}}$ stays within the sub-second threshold, and scales quite well for increasing size, performing almost as well as ALGO$_{\text{PERFECT}}$ (which is the lower bound for latency in this case) and manages to maintain a near 100% cache hit ratio, especially for smaller sampling rates. A 1-tailed t-test confirms (p-value 0.05, t-statistic $58.41 >$ required critical value 1.662) that ALGO$_{\text{DICE}}$'s speedup over ALGO$_{\text{NOSPEC}}$ is statistically significant. Another observation is a performance envelope with ALGO$_{\text{PERFECT}}$ exists – there are several constant-time overheads which could be further optimized, an opportunity for future work. Figure 4 performs the same experiment at a larger scale on CLUSTER$_{\text{CLOUD}}$, **allowing for cube exploration over the 1 billion rows (100% sampling) while maintaining a sub-second average latency – 33% faster than the baseline ALGO$_{\text{NOSPEC}}$.**

2) Sampling & Accuracy: Since *DICE* allows the user to vary the sampling rate, we present a plot of the AVERAGE ACCURACY for a sample workload, compared to results from aggregation over the full dataset. It should be noted that accuracy depends on multiple factors. First and foremost, accuracy is dependent on *skew in the data*. As described in the schema, our dataset contains a multitude of distributions across all the dimensions. Second, the selectivity of queries in the *workload* will impact the sensitivity of error. Third, the *placement* of the data is a significant contributing factor: since data is horizontally sharded across multiple nodes, sampling and aggregation of data is impacted by the uniformity of data placement. In Figure 5, we present the average accuracy for a workload at varying sampling rates over all 1B rows. For this workload, accuracy increases steadily till the 50% mark, after which the benefits of increasing the sampling taper off, slowly reaching full accuracy at the 100% sampling rate.

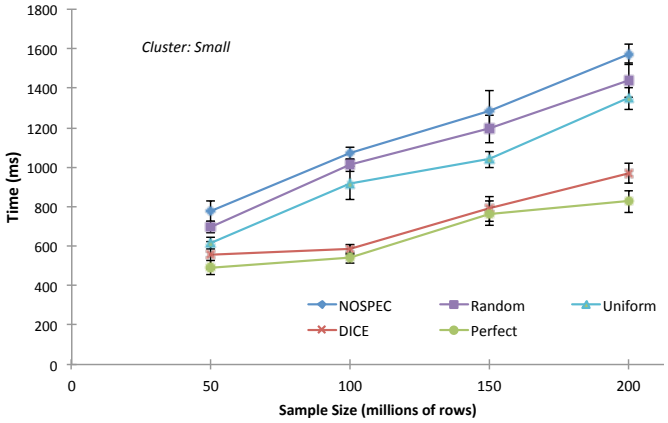

Fig. 3. Varying Size of Dataset: CLUSTER$_{\text{SMALL}}$

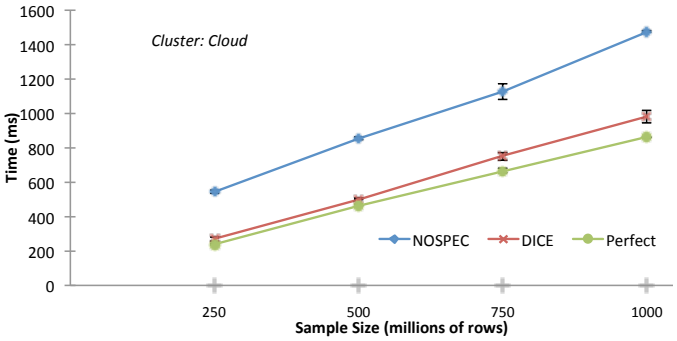

Fig. 4. Varying Size of Dataset: CLUSTER$_{\text{CLOUD}}$

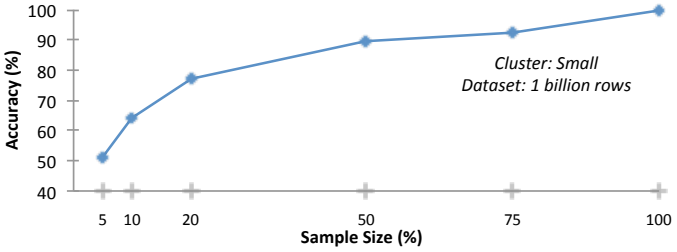

Fig. 5. Accuracy over a workload

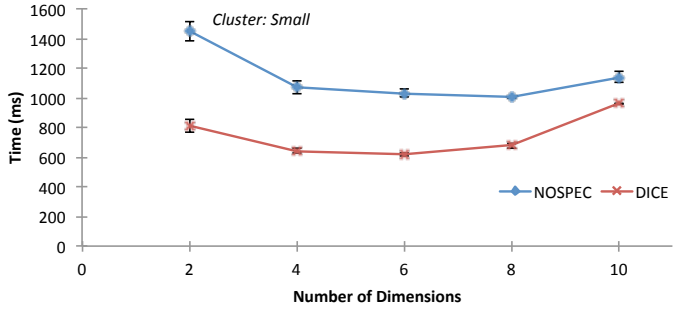

Fig. 6. Impact of Number of Dimensions

3) Number of Dimensions: Figure 6 shows how varying the number of dimensions in a query affects its execution time.

Dimensions are increased by adding new `WHERE` predicates to the query. As seen in Figure 6, execution time decreases up to a certain point and then starts increasing. The decreasing slope in the curve is caused by selectivity – as dimensions are added, less number of rows are processed, allowing for faster materialization of resultsets. After a certain point, the evaluation cost of the multiple `WHERE` clauses takes over, especially because the order of filter dimensions is not ideal.

4) Number of Slave Nodes: We vary the number of slave nodes in Figure 7, while keeping the size of the data constant at 200M rows. As expected, for all algorithms, latencies decrease as the number of nodes increases. An interesting observation is made for ALGO$_{\text{DICE}}$ however – for 4 nodes, *DICE* thrashes memory due to the amount of data involved and the number of speculative queries, which is not a problem for both ALGO$_{\text{NOSPEC}}$ (no speculation / caching) or ALGO$_{\text{PERFECT}}$ (exactly one ad-hoc query being cached).

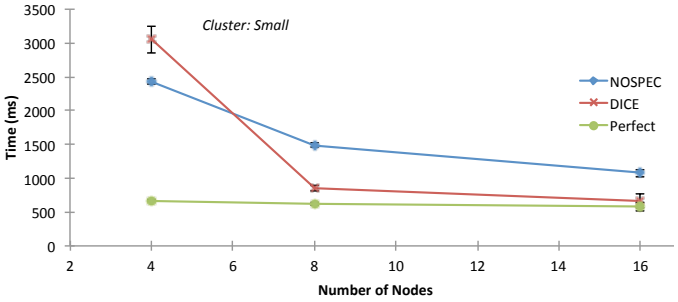

Fig. 7. Varying the Number of Slave Nodes

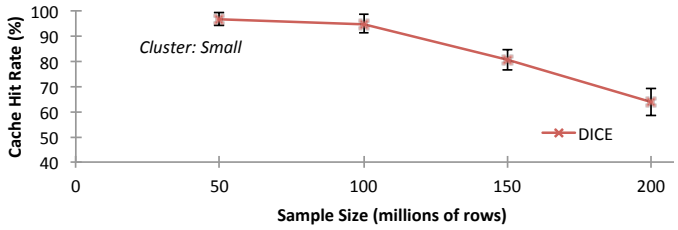

Fig. 8. Cache Hit Change with Sampling Rate Change

5) Cache Hit Variability: Since the cache hit rate is a key contributor to the average latency of a session, in Figure 8 we study how the cache hit rate varies with the sampling rate for a fixed cache size. We use the cache hit rate as a proportional measure of the prediction quality. Higher cache hits are a direct result of high quality of speculation. We achieve close to a 100% hit rate for 50 million sampled rows. As we increase the sampling rate, we see the cache hit rate decreasing nearly linearly, since the total number of speculative queries increases linearly with the sampling rate.

6) Sample Session: As an anecdotal example, we present in Figure 9 the trace of a single cube exploration session for ALGO$_{\text{NOSPEC}}$, ALGO$_{\text{DICE}}$ and ALGO$_{\text{PERFECT}}$ on CLUSTER$_{\text{SMALL}}$. The X axis depicts successive ad-hoc queries in a session. (It should be noted that while the bars are stacked together for convenience for the reader, the session for each algorithm is executed separately.) The Y axis represents AVERAGE LATENCY. Cache hit rate for ALGO$_{\text{DICE}}$ is shown as a label above the bars. The cache hit rate for the first query

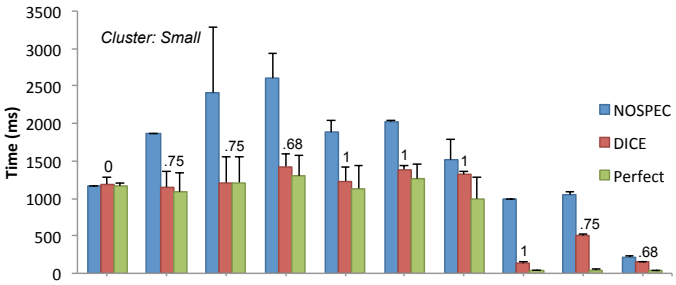

Fig. 9. Individual Latencies for Anecdotal Query Session

is 0.0, since there has been no speculation and the caches are empty. ALGO$_{\text{DICE}}$ performs almost as well as ALGO$_{\text{PERFECT}}$ with hit rates equal or closer to 1.0.

7) Impact of Various Techniques: We now study in Figure 10, the performance impact of the various algorithms and optimizations to our system on the CLUSTER$_{\text{SMALL}}$ cluster. We compare the AVERAGE LATENCY of various techniques compared to ALGO$_{\text{NOSPEC}}$. ALGO$_{\text{UNIFORM}}$ is slightly faster due to some of the speculative queries being part of the session. Including the unification optimization discussed in Section III-F reduces the number of concurrent queries, improving latency. Finally, including the locality model and cost-based prioritization of speculative queries yields ALGO$_{\text{DICE}}$, which outperforms all other methods.

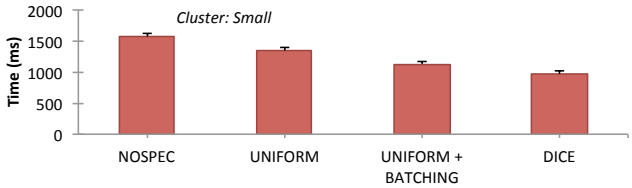

Fig. 10. Impact of Various Techniques

C. Real-world Usage and User Study

Real-world Query Logs: To evaluate the real-world efficacy of the facet model, we procured a real-world query log of ad-hoc analytical queries by real users on a production system generated HIVE data warehouse of an Internet advertising company. Considering only the aggregation queries (with the `group by` clause), the log spanned 509 queries. Amongst them 46 query sessions were detected which comprised of 116 queries i.e. 22.97% of the queries. The traversals described in the *DICE* model were found to cover 100% of the session-based queries, demonstrating that our traversal model is indeed expressive enough to allow for significant speedups (the remainder are executed traditionally, without speculation.)

User Studies: We performed a user study to compare the effectiveness *DICE* over traditional methods.The study was performed with 10 graduate students across the department who were knowledgeable in databases and data cubing, determined using a pre-test. The users were then given a pre-task tutorial on data cubing and our data model. They were then asked to explore the cube using the faceted model for 10 ad-hoc queries of their choice. They were not told if the *DICE* speculation was turned on or off (50% of the users each). After the session, the user's query session was repeated in the

other mode (speculation was turned on if it was off before and vice versa) to get comparable times. Care was taken to avoid different biases. The pre-task tutorial avoided bias against prior knowledge of data cubing and our data model. Having only 1 total task that lasted less than 10 minutes prevented fatigue bias, and the same user workload being re-run (automated) for the alternate mode avoided learning effects. The test algorithm being split equally dealt with carryover effects.

The mean and standard deviation for *Time-To-Task* for the entire query session of 10 queries using $\text{ALGO}_{\text{DICE}}$ were 47757 ms and 937 ms and for $\text{ALGO}_{\text{NOSPEC}}$ were 54506 ms and 3111 ms: **i.e. on average users queried 7 seconds faster with $\textbf{ALGO}_{\textbf{DICE}}$**. Consider null hypothesis as $\text{ALGO}_{\text{DICE}}$ *execution time to be no different than* $\text{ALGO}_{\text{NOSPEC}}$ *time* and alternate hypothesis to be that $\text{ALGO}_{\text{DICE}}$ *is faster than* $\text{ALGO}_{\text{NOSPEC}}$. The query session time which consists of query execution time, query input time and result view time is **significantly** lesser for our method, $\text{ALGO}_{\text{DICE}}$, compared with $\text{ALGO}_{\text{NOSPEC}}$ based on a 1-tailed t-test (t-statistic value of $21.77 > 1.833$ needed for a p-value of 0.05).

Speculation Noticeability: While *DICE* speedups are objectively significant, an important question for a user-involved system is: **Can users notice the difference and have a preference?** To test this, the users were asked to report which query session (i.e. with or without *DICE*) they found out to be faster. Results were unanimous: **all our users preferred the $\textbf{ALGO}_{\textbf{DICE}}$** session over $\text{ALGO}_{\text{NOSPEC}}$. Clearly, reduction in query times due to usage of $\text{ALGO}_{\text{DICE}}$ speculation is indeed noticeable to the user.

User Satisfaction: At the end of the query session, the users were asked to rate their satisfaction (10:*extremely happy*, 5:*neutral*, 1:*extremely unhappy*) for both the faceted traversal model and the overall system. For the **traversal model**, the main criteria they were asked to take into consideration were the traversals allowed under *DICE*, any extra traversals they thought it lacked, and the ease of traversal. The mean rating for the faceted model was 7.9 with a standard deviation of 1.54. Consider the null hypothesis of the faceted model ratings being equal to 5 (i.e. *neutral*) and the alternate hypothesis of the faceted model rating being greater than 5. The value of the t-statistic was found out to be 5.67 which is much greater than the critical value of 1.833 needed for a 1-tailed t-test for a p-value of 0.05 showing that the **traversal model satisfaction was statistically significantly better than random/neutral response**. Additionally, for the **overall *DICE* system**, the average *User-Satisfaction* was very high: 8.7 with a standard deviation of 0.82, summarizing our overall assertion that ***DICE* not only provides objective speedups, it also provides a significantly better experience for the end user**.

V. Related Work

Cube Exploration: While the original cube paper [19] provides for a variety of operators, *facet* traversals introduced in this paper are typical to interactions on analytics user interfaces. Work by Sarawagi et al. on mining of *interesting* regions [45] and exploration operators [46] can be easily plugged into our speculation framework. Kamber et al. [27] have discussed *metarule* exploration, and dynamic exploration on cube subsets have been discussed in [31]. Our contribution is towards **improving interactive exploration in a session context**.

Cube Materialization: Materialization strategies range from full-cube materialization over MapReduce [37] to region-specific materialization [11] to selective partial materialization. Optimization techniques exist for optimizing intra-query parallelization [3], but do not consider multiple queries as part of an interactive session.

Distributed Query Execution: Ad-hoc analysis over large datasets has been made popular with the availability of declarative query languages such as SCOPE [8], Pig [40] and Hive [52], which translate to MapReduce-oriented flows, and is not ideal for interactive workloads. Ideas such as columnar storage layouts [5], [20], [49], hierarchical execution [35], distributed database hybrids [1], online distributed aggregation [41] and main-memory engines [18] have achieved low latencies when querying over large datasets, resulting in a spurt of development activity in this area, resulting in implementations such as Drill, Impala, Tez, PivotalHD, HAWQ, Peregrine and Druid, projects that target *single query execution latency*.

Prefetching: The idea of speculative execution of queries and prefetching results has been discussed before [48], [51]. PROMISE [44] investigates the likelihood of future queries and can be used to supplant the workload-based approach in our paper. Ramachandran et al. [43] focus on the speculation of exact, non-approximate drill-down queries. Improvements in speculation quality based on ideas in these papers can be used to better prioritize and sample our speculative queries.

Online Aggregation: Online aggregation ideas proposed by Hellerstein et al. [24] and the related CONTROL [23] project which surface approximate answers are highly relevant and related work. Our system builds upon these ideas in a distributed cubing environment, combining user-directed techniques of speculative execution and sampling.

Sampling-based Estimation: There is significant prior work in using sampling for approximating query results [38]. Jin et al. [26] detail the approximation of OLAP queries using pre-summarized statistics. Wang et al. discuss [55] data placement, [56] details the computation of errors for a GROUP BY query over multitable joins, and [32] discuss a sampling-based framework to materialize cubes. BlinkDB [2] performs an offline sampling step of multiple column combinations. As mentioned before, the ideas presented in BlinkDB are orthogonal to both the faceted exploration model proposed by our work, and the speculation-based execution architecture. Strategies for stratification using prior workloads [10] and methods to increase sensitivity for low-selectivity attributes [54] have also been considered before.

Data Interaction: The proliferation of business intelligence tools that leverage visualization and interactive interfaces [4], [6], [15], [28], [57] to explore large multidimensional datasets highly motivate the need for a distributed interactive cube exploration system. Tools such as Tableau [21] translate visual

interactions into a series of SQL queries, and interactive loops correlate directly with our session-based model. As discussed in Section IV, we observe that such interactions directly correspond to facet traversals, allowing us to utilize actual workloads from such tools in our experimental evaluation. Olston et al. [39], propose the interactive analysis of web-scale data using query templates. Cetintemel et al. [7] envision a "guidance" system for interactive querying. Session-oriented sampling and speculation approaches described in our paper can significantly improve the interactivity of such a system.

VI. Conclusion and Future Work

Given the proliferation of commodity distributed infrastructure and big data analytics applications, there is a compelling need for systems that allow interactive exploration of aggregated data. As demonstrated in the experiments, *DICE* meets this need, and allows for exploration of 1-billion-tuple data cubes at sub-second latencies, significantly outperforming existing methods. The system uses a combination of three complementary strategies: a faceted cube exploration model, data sampling and speculative caching to provide interaction-level performance for the end-user.

Going forward, there are several avenues of future work. The inclusion of *interestingness* of cube groups into the exploration framework would be a very useful extension. This would bridge the gap between automated exposition of insights, and ad-hoc exploration. One possible way to include this into *DICE* is to formulate the *interestingness* of a facet, which can be mined in an initial offline step during the ingestion of a table shard, and stored in conjunction with the dataset. During the exploration phase, this can considered when prioritizing facets to speculatively execute. DICE can be trivially extended by modeling user behavior through query logs and plugging in new traversal types. We plan on modeling traversal patterns for 2 dimensional `GROUP BY`s. Further, we intend to support multi-tenancy; this would allow us to leverage caching benefits across multiple users, allowing for higher speedups. Estimated speculation time can be divided between different users based on their importance. Another possible extension is to combine methods for offline and online materialization of data cubes by identifying the fraction of the cube to fully prematerialize. A possible approach is to materialize an approximate and compressed representation [17] of the data cube, and use the online execution step to increase the quality of the answer based on the approximate model [22].

References

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *VLDB*, 2009.

[2] S. Agarwal, A. P. Iyer, A. Panda, S. Madden, B. Mozafari, and I. Stoica. Blink and It's Done: Interactive Queries on Very Large Data. *VLDB*, 2012.

[3] F. Akal, K. Böhm, and H. Schek. OLAP Query Evaluation in a Database Cluster: A Performance Study on Intra-Query Parallelism. *ADBIS*, 2002.

[4] M. Barnett, B. Chandramouli, R. DeLine, S. Drucker, D. Fisher, J. Goldstein, P. Morrison, and J. Platt. Stat!-An Interactive Analytics Environment for Big Data. *SIGMOD*, 2013.

[5] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. *CIDR*, 2005.

[6] A. Buja, D. Cook, and D. F. Swayne. Interactive High-Dimensional Data Visualization. *Computational and Graphical Statistics*, 1996.

[7] U. Çetintemel, M. Cherniack, J. DeBrabant, Y. Diao, K. Dimitriadou, A. Kalinin, O. Papaemmanouil, and S. B. Zdonik. Query Steering for Interactive Data Exploration. *CIDR*, 2013.

[8] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, et al. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *VLDB*, 2008.

[9] B. Chandramouli, J. Goldstein, R. Barga, et al. Accurate Latency Estimation in a Distributed Event Processing System. *ICDE*, 2011.

[10] S. Chaudhuri, G. Das, and V. Narasayya. Optimized Stratified Sampling for Approximate Query Processing. *TODS*, 2007.

[11] Y. Chen, A. Rau-Chaplin, et al. cgmOLAP: Efficient Parallel Generation and Querying of Terabyte Size ROLAP Data Cubes. *ICDE*, 2006.

[12] S.-J. Chun, C.-W. Chung, and S.-L. Lee. Space-Efficient Cubes for OLAP Range-Sum Queries. *DSS*, 2004.

[13] W. G. Cochran. *Sampling Techniques*. John Wiley & Sons, 2007.

[14] D. Deutch, Z. G. Ives, T. Milo, and V. Tannen. Caravan: Provisioning for What-If Analysis. *CIDR*, 2013.

[15] A. Dubrawski, M. Sabhnani, et al. Interactive Manipulation, Visualization Analysis of Large Sets of Multidimensional Time Series in Health Informatics. *INFORMS*, 2008.

[16] U. Fischer et al. Forecasting the Data Cube: A Model Configuration Advisor for Multi-Dimensional Data Sets. *CITY*, 2013.

[17] N. Friedman et al. Learning Bayesian Network Structure from Massive Datasets. *UAI*, 1999.

[18] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *TKDE*, 1992.

[19] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, et al. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1997.

[20] A. Hall et al. Processing a Trillion Cells Per Mouse Click. *VLDB*, 2012.

[21] P. Hanrahan. VizQL: A Language for Query, Analysis and Visualization. *SIGMOD*, 2006.

[22] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing Data Cubes Efficiently. *SIGMOD*, 1996.

[23] J. Hellerstein, R. Avnur, A. Chou, C. Hidber, et al. Interactive Data Analysis: The Control Project. *Computer*, 1999.

[24] J. Hellerstein et al. Online Aggregation. *SIGMOD*, 1997.

[25] T. Jäkel et al. Pack Indexing for Time-Constrained In-Memory Query Processing. *BTW*, 2013.

[26] R. Jin, L. Glimcher, C. Jermaine, and G. Agrawal. New Sampling-Based Estimators for Olap Queries. *ICDE*, 2006.

[27] M. Kamber et al. Metarule-Guided Mining of Association Rules using Data Cubes. *KDD*, 1997.

[28] D. Keim, F. Mansmann, et al. Visual Analytics: Scope and Challenges. *Visual Data Mining*, 2008.

[29] A. Kemper et al. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. *ICDE*, 2011.

[30] N. Khoussainova et al. Session-Based Browsing for More Effective Query Reuse. *SSDBM*, 2011.

[31] B. Leonhardi, B. Mitschang, R. Pulido, et al. Augmenting OLAP Exploration with Dynamic Advanced Analytics. *EDBT*, 2010.

[32] X. Li, J. Han, Z. Yin, J.-G. Lee, et al. Sampling Cube: A Framework for Statistical OLAP over Sampling Data. *SIGMOD*, 2008.

[33] X. Liu et al. A Text Cube Approach to Human, Social and Cultural Behavior in the Twitter Stream. *SBP*, 2013.

[34] S. L. Lohr. *Sampling: Design and Analysis*. Cengage Learning, 2010.

[35] S. Melnik, A. Gubarev, J. Long, et al. Dremel: Interactive Analysis of Web-Scale Datasets. *VLDB*, 2010.

[36] R. Miller. Response Time in Man-Computer Conversational Transactions. *FJCC*, 1968.

[37] A. Nandi, C. Yu, P. Bohannon, and R. Ramakrishnan. Distributed Cube Materialization on Holistic Measures. *ICDE*, 2011.

[38] N. Ntarmos, P. Triantafillou, et al. Statistical Structures for Internet-Scale Data Management. *VLDB*, 2009.

[39] C. Olston, E. Bortnikov, K. Elmeleegy, F. Junqueira, and B. Reed. Interactive Analysis of Web-Scale Data. *CIDR*, 2009.

[40] C. Olston, B. Reed, et al. Pig Latin: A Not-So-Foreign Language for Data Processing. *SIGMOD*, 2008.

[41] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online Aggregation for Large Mapreduce Jobs. *PVLDB*, 2011.

[42] C. Raïssi et al. Computing Closed Skycubes. *VLDB*, 2010.

[43] K. Ramachandran, B. Shah, and V. V. Raghavan. Dynamic Pre-Fetching of Views Based on User-Access Patterns in an OLAP System. *SIGMOD*, 2005.
[44] C. Sapia. PROMISE: Predicting Query Behavior to Enable Predictive Caching Strategies for OLAP Systems. *DaWaK*, 2000.
[45] S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-Driven Exploration of OLAP Data Cubes. *EDBT*, 1998.
[46] S. Sarawagi and G. Sathe. i3: Intelligent, Interactive Investigation of OLAP Data Cubes. *SIGMOD*, 2000.
[47] B. Shneiderman. Response Time and Display Rate in Human Performance with Computers. *CSUR*, 1984.
[48] A. Smith. Sequentiality and Prefetching. *TODS*, 1978.
[49] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, et al. C-Store: A Column-Oriented DBMS. *VLDB*, 2005.
[50] F. Tao et al. EventCube: Multi-Dimensional Search and Mining of Structured and Text Data. *KDD*, 2013.
[51] F. Tauheed et al. SCOUT: Prefetching for Latent Structure Following Queries. *VLDB*, 2012.
[52] A. Thusoo, J. Sarma, N. Jain, et al. Hive-A Petabyte Scale Data Warehouse using Hadoop. *ICDE*, 2010.
[53] D. Tunkelang. Faceted Search. *Synthesis Lectures on Information Concepts, Retrieval, and Services*, 2009.
[54] F. Wang and G. Agrawal. Effective Stratification for Low Selectivity Queries on Deep Web Data Sources. *CIKM*, 2011.
[55] Y. Wang, S. Parthasarathy, and P. Sadayappan. Stratification Driven Placement of Complex Data: A Framework for Distributed Data Analytics. *ICDE*, 2013.
[56] F. Xu, C. Jermaine, and A. Dobra. Confidence Bounds for Sampling-based Group by Estimates. *TODS*, 2008.
[57] J. Yi et al. Toward a Deeper Understanding of the Role of Interaction in Information Visualization. *VCG*, 2007.

APPENDIX

We build upon stratified sampling and post-stratification [13] for our sampling framework. Error bounds for aggregation queries are based on the variance (across samples) of the measure for each cube group. As an initial step, we combine the variance for the same group across multiple queries, after which the variances across multiple groups are combined to give an error estimate for the entire query.

TABLE I
LIST OF NOTATIONS USED IN SECTION II-C

Symbol	Explanation
s_h^2	variance of the group h
n_h	number of tuples in the group h in the sample
n_{hi}	number of tuples belonging to the group h from the i^{th} query
n	total number of tuples in the sample
m_{hi}	mean of the group h from the i^{th} query
m_h	mean of the group h from all the queries
v_{hi}	variance of the group h from the i^{th} query
p	proportion of tuples selected by the where clause
$\hat{V}[\hat{\theta}]$	variance of the estimator for the parameter θ
H	number of groups in the union of all the queries
N_h	number of tuples in group h in the dataset
N	number of tuples in the dataset

Combining variances within groups: In order to deliver results at higher sampling rates, *DICE* runs the same query on multiple randomly chosen shards on multiple nodes. This results in the same cube group being possibly obtained from the multiple table shards. Hence, the statistics for the same group from these multiple queries need to be combined together. While combining the AVG, SUM and COUNT is straight forward, we present a technique for combining variances as

$$s_h^2 = \frac{1}{n_h - 1}\Big(\sum_{i=1}^{numQ} n_{hi}(m_{hi} - m_h)^2) + \sum_i (n_{hi} - 1)v_{hi}\Big) \quad (2)$$

where $numQ$ is the number of queries that a query needs to be replicated to. Thus, we get the requisite statistics for a combined group across all the replicated queries.

Continuing our motivating example, the faceted representation of the query is $f(rack, hour : 6, datacenter : EU)$ with the measure AVG and measure dimension iops . We append the COUNT and VARIANCE measures to the queries since they are needed as given in Equation (2) to combine variances for the same group across multiple queries. Assume the query is run on a single shard on 2 nodes and result into a sampling rate of 10%, returning us groups and the corresponding measures from the two queries respectively as:

{[rack:1,hour:6,datacenter:EU,AVG:10,COUNT:5,VAR:4],
[rack:2,hour:6,datacenter:EU,AVG:12,COUNT:6,VAR:2]} &
{[rack:1,hour:6,datacenter:EU,AVG:5,COUNT:8,VAR:1],
[rack:2,hour:6,datacenter:EU,AVG:6,COUNT:7,VAR:2]}.

Plugging in the values from above into (2), we get the variance for the combined group [rack:1,hour:6,datacenter:EU] as $s_1^2 = 8.32$ and for [rack:2,hour:6,datacenter:EU] as $s_2^2 = 11.52$.

Combining variances across groups: From the variances of each of the combined groups, we can get an error estimate for the combination of all of these groups i.e. the combined result set. We consider three algebraic measures SUM, AVG and COUNT. From the standard sampling theory, the variance of the estimator for the measure SUM can be given as:

$$\hat{V}[\hat{t}] = \sum_{h=1}^{H} N_h^2 (1 - \frac{n_h}{N_h}) \frac{\hat{s_h}^2}{n_h} \quad (3)$$

The variance of the estimator for the measure AVG can be obtained by dividing the above value by N^2.

For the measure COUNT, we can use the proportion estimator since the where clause acts as the indicator function and thus the variance of the estimator for COUNT can be given as:

$$\hat{V}[\hat{p}] = (1 - \frac{n}{N}) \frac{\hat{p}(1 - \hat{p})}{n - 1} \quad (4)$$

The above formulae cannot be used as they are since we cannot know the value of N_h without accessing the entire data. We resolve this issue by estimating $\frac{N_h}{N}$ by $\frac{n_h}{n}$ and $\frac{n_h}{N_h}$ by the sampling rate which resulting into an unbiased estimator. Holistic measures can be handled as described in [34].

Again plugging in the values we get, $\hat{y} = 6.92 * 13/26 + 8.77 * 13/26 = 7.85$ and $\hat{V}[\hat{t}] = (\frac{13}{26})^2 * (1 - 0.1) * (8.32/13 + 11.52/13) = 0.35$

The error will be given by

$$\frac{ConfidenceInterval}{2 * Estimate} = z_{\frac{\alpha}{2}} * \frac{\sqrt{V(\hat{\theta})}}{\hat{\theta}} \quad (5)$$

where $\hat{\theta}$ is the estimate of the measure parameter and $V(\hat{\theta})$ is the variance of the estimate.

Thus, for a confidence of 95%, we can get the standard error for the query as $\frac{1.96 * \sqrt{0.35}}{7.85} = 0.15$ resulting into an accuracy of 85%.