Scuba: Diving into Data at Facebook

Lior Abraham* John Allen Oleksandr Barykin
Vinayak Borkar Bhuwan Chopra Ciprian Gerea
Daniel Merl Josh Metzler David Reiss
Subbu Subramanian Janet L. Wiener Okay Zed
Facebook, Inc. Menlo Park, CA

ABSTRACT

Facebook takes performance monitoring seriously. Performance issues can impact over one billion users so we track thousands of servers, hundreds of PB of daily network traffic, hundreds of daily code changes, and many other metrics. We require latencies of under a minute from events occuring (a client request on a phone, a bug report filed, a code change checked in) to graphs showing those events on developers' monitors.

Scuba is the data management system Facebook uses for most real-time analysis. Scuba is a fast, scalable, distributed, in-memory database built at Facebook. It currently ingests millions of rows (events) per second and expires data at the same rate. Scuba stores data completely in memory on hundreds of servers each with 144 GB RAM. To process each query, Scuba aggregates data from all servers. Scuba processes almost a million queries per day. Scuba is used extensively for interactive, ad hoc, analysis queries that run in under a second over live data. In addition, Scuba is the workhorse behind Facebook's code regression analysis, bug report monitoring, ads revenue monitoring, and performance debugging.

1. INTRODUCTION

At Facebook, whether we are diagnosing a performance regression or measuring the impact of an infrastructure change, we want data and we want it fast. The Facebook infrastructure team relies on real-time instrumentation to ensure the site is always running smoothly. Our needs include very short latencies (typically under a minute) between events occuring on the web servers running Facebook to those events appearing in the graphs produced by queries.

Flexibility and speed in querying data is critical for diagnosing any issues quickly. Identifying the root cause for a issue is often difficult due to the complex dependencies between subsystems at Facebook. Yet if any issues are not fixed within minutes to a few hours, Facebook's one billion users become unhappy and that is bad for Facebook.

*Lior Abraham, John Allen, and Okay Zed were key early contributors to Scuba who have left Facebook. Vinayak Borkar is a Facebook graduate fellow from U.C. Irvine.

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 39th International Conference on Very Large Data Bases, August 26th - 30th 2013, Riva del Garda, Trento, Italy.
Proceedings of the VLDB Endowment, Vol. 6, No. 11
Copyright 2013 VLDB Endowment 2150-8097/13/09... $ 10.00.

Originally, we relied on pre-aggregated graphs and a carefully managed, hand-coded, set of scripts over a MySQL database of performance data. By 2011, that solution became too rigid and slow. It could not keep up with the growing data ingestion and query rates. Other query systems within Facebook, such as Hive [20] and Peregrine [13], query data that is written to HDFS with a long (typically one day) latency before data is made available to queries and queries themselves take minutes to run.

Therefore, we built Scuba, a fast, scalable, in-memory database. Scuba is a significant evolution in the way we collect and analyze data from the variety of systems that keep the site running every day. We now use Scuba for most real-time, ad-hoc analysis of arbitrary data. We compare Scuba to other data management systems later in the paper, but we know of no other system that both ingests data as fast and runs complex queries as fast as Scuba.

Today, Scuba runs on hundreds of servers each with 144 GB RAM in a shared-nothing cluster. It stores around 70 TB of compressed data for over 1000 tables in memory, distributed by partitioning each table randomly across all of the servers. Scuba ingests millions of rows per second. Since Scuba is memory-bound, it expires data at the same rate. To constrain the amount of data, Scuba allows rows to specify an optional *sample_rate*, which indicates that Scuba contains only a fraction (often 1 in 100 to 1 in 1,000,000) of the original events. This sampling is necessary for events like Facebook client requests, which occur millions of times per second. Sampling may be either uniform or based on some key, such as user id. Scuba compensates for the sample_rate when computing aggregates.

In addition to a SQL query interface (for a subset of SQL including grouping and aggregations but not joins), Scuba provides a GUI that produces time series graphs, pie charts, distributions of column values, and a dozen other visualizations of data besides tables with text. Figure 1 shows a time series graph with a week over week comparison of page traffic in the Scuba GUI. In the backend, an aggregation tree distributes each query to every server and then gathers the results to send back to the client.

Although Scuba was built to support performance analysis, it soon became the system of choice to execute exploratory queries over other time-sensitive data. Many teams at Facebook use Scuba:

- Mobile development teams use Scuba to track which fractions of users are running different mobile devices, operating systems, and versions of the Facebook app.

- Ads uses Scuba to monitor changes in ad impressions, clicks, and revenue. When a drop occurs, they can narrow it down quickly to a particular country, ad type, or server cluster and determine the root issue.

Figure 1: Scuba's web user interface. The query shown on the left side generates a time series graph with a week over week comparison of three columns related to Facebook page dispatches. The dotted lines represent the same days one week earlier. It is very easy to see daily and weekly cyclical behavior with these graphs.

- Site reliability watches server errors by using Scuba. When a spike occurs, they can pinpoint whether it is due to a bug in a particular endpoint, a service in a particular datacenter or server cluster, or a physical issue with part of a datacenter.

- Bug report monitoring runs thousands of queries every hour to look for spikes in the number of bugs reported by Facebook users, grouped by dozens of demographic dimensions (location, age, friend count, etc).

In general, users start by asking high-level aggregate queries to identify interesting phenomena in their data and then dive deeper (hence the name Scuba) to find base data points of interest. In all of the above cases, being able to break down the data along multiple dimensions in an ad hoc manner is crucial.

Scuba is also the engine that underlies Facebook's code regression analysis tool, bug report monitoring tool, real-time post content monitoring tool (e.g., how many Facebook posts mention the movie "Argo"?), and many other tools. The key feature of Scuba is that queries take less than a second to execute, even when scanning hundreds of GB of data, and results are usually live over events that occurred a minute ago.

In Section 2, we describe some of the use cases supported by Scuba at Facebook, including performance monitoring, trend spotting, and pattern mining. A detailed description of Scuba's architecture, storage, and query capabilities is in Section 3. We present a simple analytical model of Scuba's query execution in Section 4. In Section 5, we evaluate Scuba experimentally. We study its speedup and scaleup properties with real data and queries. In Section 6, we compare Scuba to related work. We conclude in Section 7 with a list of ways that Scuba differs from most other database systems. We find that these differences make Scuba suit our use cases at Facebook.

2. SCUBA USE CASES

Scuba currently stores over 1000 tables. In this section, we describe a few representative use cases of Scuba.

2.1 Performance Monitoring

The original and most common use of Scuba is for real-time performance monitoring. Julie monitors the performance of facebook.com. She starts by looking at a Scuba dashboard of tens of graphs showing CPU load on servers; numbers of cache requests, hits, and misses; network throughput; and many other metrics. These graphs compare performance week over week, as in Figure 1, or before and after a big code change. Whenever she finds a significant performance difference, she then drills down through different columns (often including stack traces), refining the query until she can pin the difference to a particular block of code and fill out an urgent bug report.

Julie's dashboard runs canned queries over data that is no more than seconds old. Performance bugs can often be spotted (and fixed!) within minutes to hours of their introduction — and while they are still being tested on a fraction of the site. Alerts on spikes in these performance graphs automate much of her initial monitoring. Logging and importing data in real-time over all of Facebook's servers would be too expensive; Julie's table contains *samples* of about 1 in 10,000 events (but it varies for different tables and types of events). Scuba records the sampling rate and compensates for it.

2.2 Trend Analysis

Another time sensitive use of Scuba is trend spotting. Eric looks for trends in data content. He extracts sets of words from user posts and looks for spikes in word frequencies over time and across many dimensions: country, age, gender, etc. Like Julie, Eric analyzes seconds-old data. He built a tool for Facebook's communications

team to graph how many posts mention current phrases. This tool was used live before the Oscar awards, for example, to see how many posts in the last hour contained the names of contending movies. Unlike Julie, Eric usually writes new custom queries as he tries out new ideas for trend analysis. He also writes custom Javascript functions to calculate statistics about the data, such as co-variance between integer columns.

2.3 Pattern Mining

Product analysis and pattern mining is a third use case of Scuba. Unlike Julie and Eric, Bob is not a software engineer. He is a product specialist, analyzing how different Facebook users respond to changes in the website or mobile applications. He looks for patterns based on different dimensions, such as location and age of user, product (device, OS, and build), and keywords in bug reports, without knowing which dimensions might matter. Bob looks at data in many tables without knowing exactly how it was logged or which columns might exist. He looks for whatever patterns he can find. He uses Scuba in order to run rollup queries in milliseconds, not the minutes they take in Hive.

3. SCUBA OVERVIEW

In this section, we provide an architectural overview of Scuba. As shown in Figure 2, Scuba's storage engine consists of many independent servers, each divided into logical units of storage called "Leaf Nodes" (or leaves). The number of cpu cores determines the number of leaves, currently 8 per server. Each leaf contains a partition of data for most tables and all queries go to all leaves, as described below. We now describe the data model of Scuba.

3.1 Data model

Scuba provides a standard table model to its users. Each table has rows containing columns of data of four possible types:

1. **Integers**: Integers are used in aggregations, comparisons, and grouping. **Timestamps** are also stored as integers.
2. **Strings**: Strings are used for comparisons and grouping.
3. **Sets of Strings**: Sets of strings are used to represent, say, words in a Facebook post or the set of features (such as graph search, news feed redesign, etc.) that are true for a given user.
4. **Vectors of Strings**: Vectors of strings are ordered and are mostly used for stack traces, where the order corresponds to the stack level.

Note that floats are not supported; aggregation over floating point numbers on many leaves can cause too many errors in accuracy. Instead, Scuba recommends that users choose the number of decimal places they care about, say 5, and store $trunc(X * 10^5)$ as an integer instead of the floating point X.

Since Scuba captures data about time-varying phenomena, every row has a mandatory timestamp. These timestamps represent the time of the actual event (client request, bug report, post, etc.). Any table may have an arbitrary number of columns of one or more types. All tables contain integers and strings; only some tables contain sets or vectors of strings.

3.2 Data layout

Figure 3 shows the compression methods Scuba uses for each data type. Integers that can be represented naturally using N bytes - 1 bit are directly encoded using N bytes. Dictionary encoding means that each string is stored once in a dictionary and its index in the dictionary (an integer) is stored in the row. String columns can be stored compressed or uncompressed, depending on how many distinct values there are. For compressed string columns, each index is stored using the number of bits necessary to represent the maximum index. Uncompressed columns store the raw string and its length. For sets of strings, the indexes are sorted and delta encoded and then each index is Fibonacci encoded. (Fibonacci encoding uses a variable number of bits.) The encoded indexes are stored consecutively in the row. For vectors, there is a 2 byte $count$ of the strings and a 1 byte $size$ for the number of bits in the maximum dictionary index. Each index is then stored consecutively in the row using $size$ number of bits. All dictionaries are local to each leaf and separate for each column. Compressing the data reduced its volume by over a factor of 6 (it varies per table) as compared to storing 8 byte integers and raw strings in every column.

Scuba currently stores the table in row order, since its original use cases accessed most columns of the table in every query. Given that Scubas use cases have become more general since then, we are now exploring column-oriented storage layouts. Others [18, 8] have shown that column stores generally get better compression and better cache locality.

Scuba's data model differs from the standard relational model in two key ways. First, there is no create table statement; a table is created on each leaf node whenever the leaf first receives data for it. Since the leaves receive data at different times, the table may exist only on some leaves and may have a different schema on each leaf. Scuba presents a single table image to its users, however, despite the different schemas, by treating any missing columns as null values. Second, the columns within the table's rows may be sparsely populated; it is common for there to be 2 or 3 different row schemas within a table or for a column to change its type over time (usually to achieve better compression). Together, these two differences let Scuba adapt tables to the needs of its users without any complex schema evolution commands or workflows. Such adaptation is one of Scuba's strengths.

3.3 Data ingestion, distribution, and lifetime

Figure 2 shows the ingestion path of data into Scuba. Facebook's code base contains logging calls to import data into Scuba. As events occur, these calls are executed and (after weeding out entries based on an optional sampling rate) log entries are written to Scribe. Scribe is an open-source distributed messaging system for collecting, aggregating, and delivering high volumes of log data with low latency. It was developed by and is used extensively at Facebook [5]. A tailer process then subscribes to the Scribe categories intended for Scuba and sends each batch of new rows to Scuba via Scuba's Thrift API. (Thrift [7] is a software library that implements cross-language RPC communication for any interfaces defined using it.) These incoming rows completely describe themselves, including their schema.

For each batch of incoming rows, Scuba chooses two leaves at random and sends the batch to the leaf with more free memory. The rows for each table thus end up partitioned randomly across all leaves in the cluster. There are no indexes over any table, although the rows in each batch have timestamps in a very short time window. (These time windows may overlap between batches, however, since data is generated on many servers.)

The leaf receiving the batch stores a gzip compressed copy of the batch file to disk for persistence. It then reads the data for the new rows, compresses each column, and adds the rows to the table in memory. The elapsed time from an event occuring until it is stored in memory and available for user queries is usually within a minute.

Memory (not cpu) is the scarce resoure in Scuba. We currently

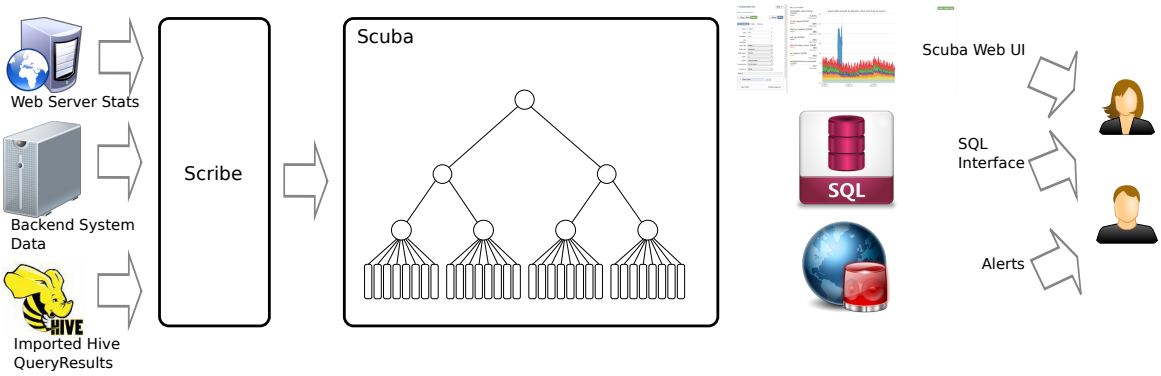

Figure 2: Scuba system architecture: from data ingestion on the left to user queries on the right.

Data Type	Compression type	Representation in row
Integer	Variable length	1-8 bytes
String	Dictionary	Index, uses number of bits for max dictionary index
String (alternate)	Uncompressed	4 bytes length + actual string
Sets of String	Dictionary	Fibonacci encoding of deltas between sorted indexes
Vectors of String	Dictionary	2 bytes count + 1 byte index size + each index

Figure 3: Data types and compression methods in Scuba.

add new machines every 2-3 weeks to keep up with the increase in tables and data. Since Scuba is intended for analysis of today's data, possibly with week over week comparisons, we delete old data at the same rate we receive new data in order to constrain table size. Data can be pruned for one of two reasons:

- **Age**: The row, as determined by its timestamp, is too old.
- **Space**: The table has exceeded its space limit and this row is one of the oldest in the table.

Most tables have default constraints of 30 days[1] and 100 GB, although there are higher limits for high volume tables like fbflow, which holds network traffic data, and ads_metrics, which records revenue-generating data like ad clicks and impressions. Every 15 minutes, a cron job evicts data that has exceeded its age limit. If the table exceeds its space limit, the oldest data for the table is evicted until it is under its limit.

In order to keep some data around longer than the space limits allow, Scuba also provides *subsampling* of data. In this case, a uniform fraction of the rows older than a certain age are kept and the remainder deleted. In the future, we would like to explore more sophisticated forms of sampling, such as stratified sampling, that might choose a more representative set of rows.

3.4 Query model

Scuba provides three query interfaces, as illustrated on the right in Figure 2.

- The web-based interface shown in Figure 1 allows users to issue form-based queries and choose one of about a dozen visualizations, including tables, time series graphs, pie charts, stacked area graphs, and many more. Figures 4, 5, and 6 show three more of these visualizations. Switching between visualizations takes only seconds.
- The command line interface accepts queries in SQL.
- The Thrift-based API allows queries from application code in PHP, C++, Java, Python, Javascript, and other languages.

The SQL interface and GUI themselves use the Thrift interface to send queries to Scuba's backend. Scripts can also issue queries using either the SQL or Thrift interfaces. Finally, we provide a mechanism to execute user-defined functions written in Javascript.

Scuba queries have the expressive power of the following SQL query:

```
SELECT column, column, ...,
  aggregate(column), aggregate(column), ...
FROM table
WHERE time >= min-timestamp
  AND time <= max-timestamp
 [AND condition ...]
GROUP BY column, column, ...
ORDER BY aggregate(column)
LIMIT number
```

The aggregate functions supported include the traditional count, min, max, sum, and average functions, as well as other useful functions such as sum/minute, percentiles, and histograms. The WHERE clause must contain a time range, although other conditions are optional. The LIMIT clause defaults to 100,000 rows to avoid memory issues in grouping and rendering problems at the client. The GROUP BY and ORDER BY clauses are wholly optional.

Any comparison to a string may include a regular expression. Conditions on sets of strings are *set-column includes any/all/none of string-list* and *set-column is empty*. Conditions on vectors of strings are *vector-column includes any/all/none/start/end/within of string-list*. The order of strings in the *string-list* is significant.

Joins are not supported in Scuba. When combining data from multiple sources is necessary, joining is usually done before importing the data into Scuba.

[1]Archival data is not stored in Scuba. Facebook uses other, disk-based, systems for long-term retention.

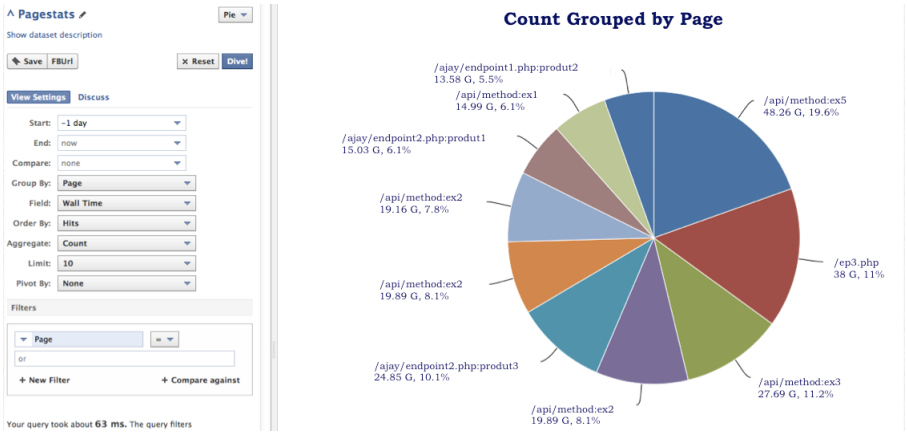

Figure 4: Scuba's pie chart visualization: this query returns the 10 pages with the highest counts for the last day. The page names have been scrubbed.

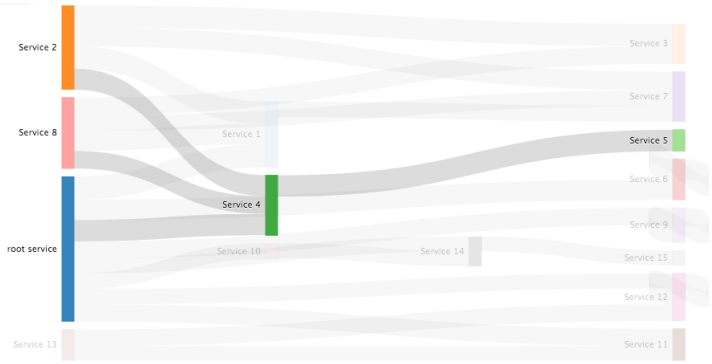

Figure 5: Scuba's sankey view: network traffic flow between services. In sankey diagrams the height of the bars is proportional to the quantity of the flow. In this example, the user is drilling down into Service 4.

3.5 Query execution

Figure 7 shows a step-by-step break down of how Scuba executes a user query. All communication between aggregators and leaves is via Thrift.

1. A client locates one of several Root Aggregators in the Scuba cluster and sends a query. The Root Aggregator receives the query, parses it, and validates it to make sure the query is well-formed.

2. The Root Aggregator identifies four other machines in the cluster to act as Intermediate Aggregators at the next level down. This step creates a fanout of five (four other machines plus itself). The Root Aggregator replaces any average functions with a sum and a count (so aggregation can be computed at the end) and sends the query to the Intermediate Aggregators.

3. The Intermediate Aggregators create further fanouts of five and propagate the query until the (only) Leaf Aggregator on each machine receives the query.

4. The Leaf Aggregator sends the query to each Leaf Server on the machine to process in parallel.

5. The Leaf Aggregator collects the results from each Leaf Server and aggregates them. It applies any sorting and limit constraints, however, for its $limit$, it uses max($5 * limit$, 100), hoping that all groups in the final top $limit$ are passed all the way up the tree. The Leaf Aggregator also collects statistics on whether each Leaf contained the table, how many rows it processed, and how many rows satisfied the conditions and contributed to the result. The Leaf Aggregator returns its result and statistics to the Intermediate Aggregator that called it.

6. Each Intermediate Aggregator consolidates the partial results it receives and propagates them up the aggregation tree.

7. The Root Aggregator computes the final result, including any averages and percentiles, and applies any final sorting and limit constraints.

8. The Root Aggregator returns the result to the waiting client, usually within a few hundred milliseconds.

Several details about how the Leaf Server processes the query are important. First, each Leaf Server may contain zero or more partitions of the table, depending on how big the table is and how long the table has existed. Very new or very small tables may be stored on only a few or a few hundred leaves out of the thousands of leaves in our cluster.

Second, the Leaf Server must scan the rows in every partition of the table whose time range overlaps the time range of the query.

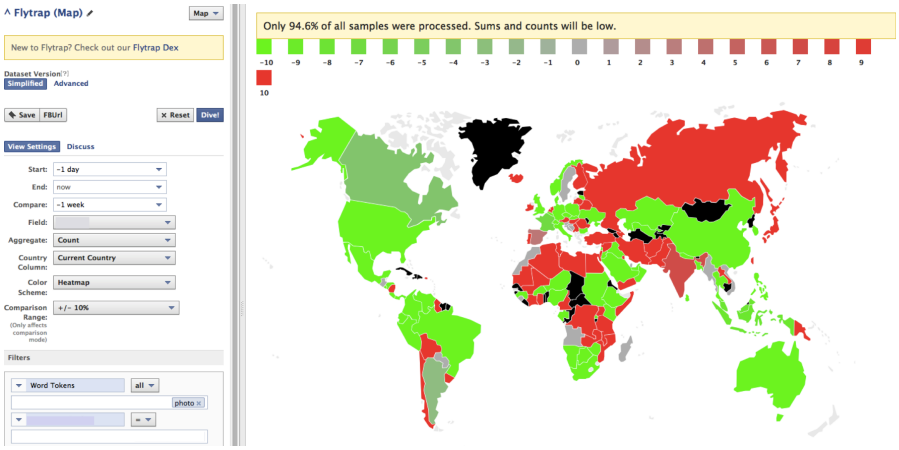

Figure 6: Scuba's world map view: the colors indicate the percentage of change in bug reports about photos over the last hour. Note that when this query was executed, some leaves were unavailable. However, the missing leaves are likely to affect the data from today and a week ago equally, so the percentage change numbers will be accurate.

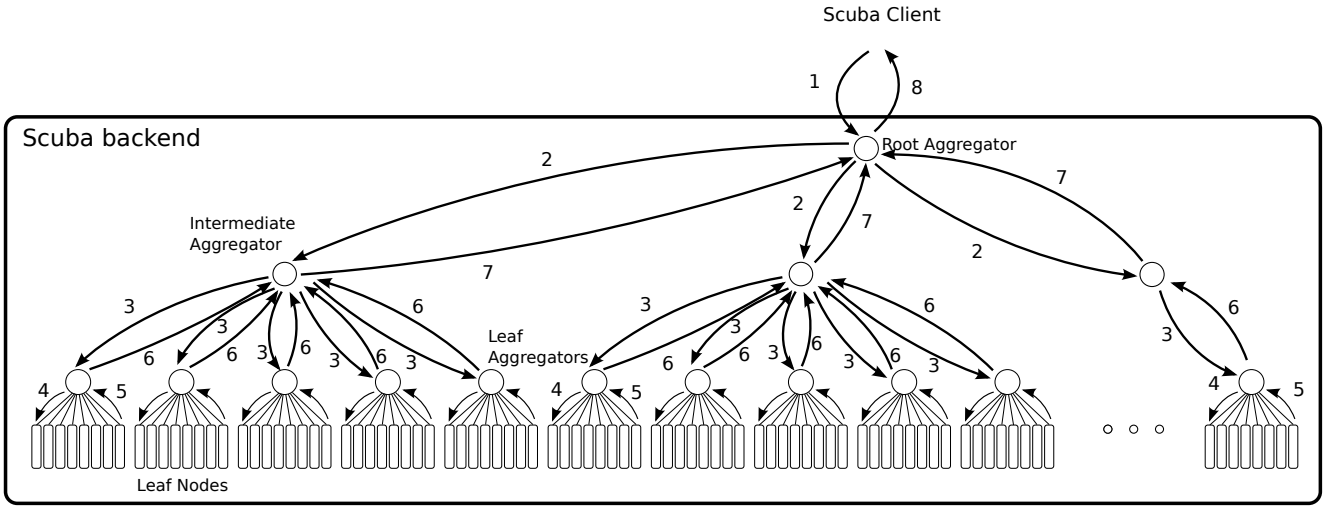

Figure 7: Step-wise breakdown of executing a Scuba query.

Non-overlapping partitions are skipped. These time ranges for the partitions are the only form of "index" that Scuba has.

Third, the Leaf Server optimizes regular expression matching per string column predicate. Whenever the string column uses a dictionary (which roughly corresponds to whenever string values are likely to be repeated in the column) the Leaf Server maintains a per-query cache of the results of matching the expression against each string value. This cache is indexed by the dictionary index for the string.

Fourth, if an Aggregator or Leaf Server does not respond within a timeout window (such as 10 ms), its results are omitted from the final computation. We have found that in practice this approach works well because of the large number of samples involved in answering Scuba queries. A few missing pieces of data do not adversely impact average and percentile computations, The much lower response time achieved by ignoring a few leaves compensates for the missing data. In addition, Scuba maintains and checks an independent service for the count of rows expected per table per query. It then uses this count to estimate the fraction of data that is actually missing. If the fraction is 99.5% or less, Scuba prints a warning in the GUI. We are also working on repeating the query automatically for those clients that require 100% accurate results.

Finally, multiple Leaf Servers and one Aggregator Server run on each physical machine. Each machine can provide the Aggregator Server at any level of the aggregation tree.

Currently, the aggregation tree fanout is five. We experiemented with fanouts of 2,4,5,6 and empirically found five to produce the best response times. Independently, Rosen, et al's [14] theoretical analysis of aggregation trees showed that the fan out of the aggregation tree with the least response time is a constant, independent of the number of leaf nodes in the tree. Their empirical analysis also showed that a fanout of five led to minimum response times in their system.

Parameter	Description
D	Number of leaf nodes per machine
N	Number of machines in the Scuba cluster
F	Fan out at each aggregator
T_D	Time to distribute a query from an aggregator to its children (transfer time)
T_G	Time to gather query results at an aggregator (transfer time)
T_A	Time to update the local aggregation state at an aggregator with results from one child
T_S	Time to scan data at a leaf node and compute local aggregated results

Figure 8: Input parameters to model Scuba's performance.

4. PERFORMANCE MODEL OF SCUBA

As described in the previous section, Scuba uses an aggregation tree to execute user queries. In this section, we construct a simple analytical model of the system. This model helped us understand the performance characteristics of a single query.

Figure 8 describes the parameters of this model. We first describe the parameters of the aggregation tree used for query processing. The expected number of levels L in an aggregation tree with fanout F and N machines is shown in Equation 1.

$$L = \lceil log_F(N) \rceil \tag{1}$$

Most of the Aggregators will have an actual fanout of F. However, when the number of machines in the Scuba cluster, N, is not a perfect power of F, the fanout of the lowest level of Intermediate Aggregators is smaller than F. Equation 2 shows the fanout at the penultimate level in the aggregation tree.

$$R = \frac{N}{F^{L-1}} \tag{2}$$

F^{L-1} represents the number of Aggregators at level $L-1$ in the aggregation tree. Therefore, R represents the fanout at the last level of the tree to reach all N Leaf Aggregators from each of the F^{L-1} Intermediate Aggregators. (Recall that each machine has one Leaf Aggregator.)

We can now describe the fanout at level L in Equation 3.

$$F_L = \begin{cases} F & \text{if } L > 1 \\ R & \text{if } L = 1 \\ D & \text{if } L = 0 \end{cases} \tag{3}$$

The last case in Equation (3) describes the fanout D from a Leaf Aggregator to the Leaf Nodes on the same machine.

Each Aggregator in the tree performs the following steps:

1. Distribute the query to each child Aggregator and wait for results.
2. Incorporate results from each child as they are received. When all children have responded or the query times out, stop waiting.
3. Return consolidated aggregation results and response statistics to the caller.

Note that once queries are forwarded to its children by an Aggregator, the children proceed in parallel. So the total response time of a query computed using an aggregation tree with L levels is represented by T_L in Equation 4.

$$T_L = \begin{cases} T_D + T_G + T_{L-1} + T_A F_L & \text{if } L > 0 \\ T_S & \text{if } L = 0 \end{cases} \tag{4}$$

Expanding Equation (4) and incorporating Equation (3) produces Equation 5.

$$T_L = T_S + L(T_D + T_G) + F(L-1)T_A + R + D \tag{5}$$

T_L is thus the predicted time for any query (in the absense of query contention). We validated (and refined) this model by plugging in actual values for T_A, T_D, T_G, and T_S in each query and comparing them to our experimental results. However, we do not present the comparison here.

5. EXPERIMENTAL EVALUATION

In this section, we present the results of experiments to measure Scuba's speed up and scale up on a test cluster of 160 machines.

5.1 Experimental setup

The machines in our test cluster are Intel Xeon E5-2660 2.20 GHz machines with 144 GB of memory [19]. There are four racks of 40 machines each, connected with 10G Ethernet. The operating system is CentOS release 5.2.

For these experiments, we vary the number of machines in the cluster from 10 to 160. In every experiment, each machine has 8 Leaf Nodes and 1 Aggregator. Each Aggregator always serves as a Leaf Aggregator and can additionally be an Intermediate and Root Aggregator.

5.2 Experimental queries and data

The experiments are meant to isolate the various parameters of the model in Section 4. Therefore, we use 1 table containing 29 hours worth of real data (copied from our production cluster) and 2 very simple queries. The total amount of data in the table is about 1.2 TB. (Except where stated otherwise, each leaf has 1 GB; there are 8 leaves per machine and 160 machines. ($1 * 8 * 160 = 1280$) We ran two different queries.

```
SELECT count(*), SUM(column1) as sum1,
   SUM(column2) as sum2
FROM mytable
WHERE time >= now()-3*3600
```

The first query, shown above in SQL, isolates scan time at the leaves. It scans over 3, 6, or all 29 hours of the data (depending on the constants), computes 3 aggregation metrics, and passes (only) those 3 numbers (and query statistics) up the aggregation tree.

```
SELECT count(*), sum(column1) as sum1,
   service,
   (time - now())/60*60 + now() as minute,
FROM mytable
WHERE time >= now()-3*3600
   and time <= now()
GROUP BY service, minute
ORDER BY sum1 DESC
LIMIT 1800
```

The second query is more complicated, especially in SQL, as shown above. This query produces a time series graph over the last

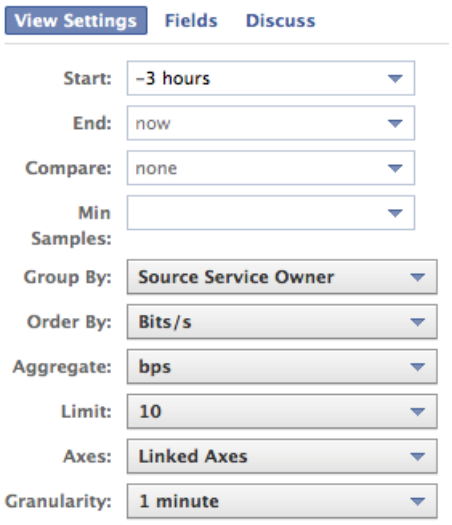

Figure 9: Time series query in Scuba GUI

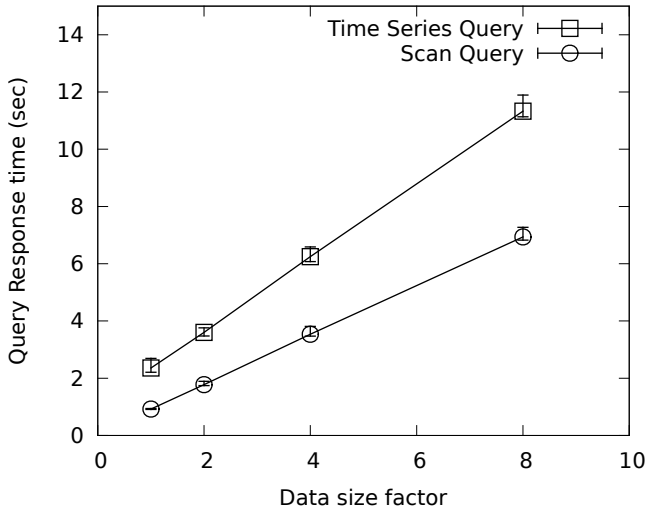

Figure 10: Measuring speed up as the amount of data increases.

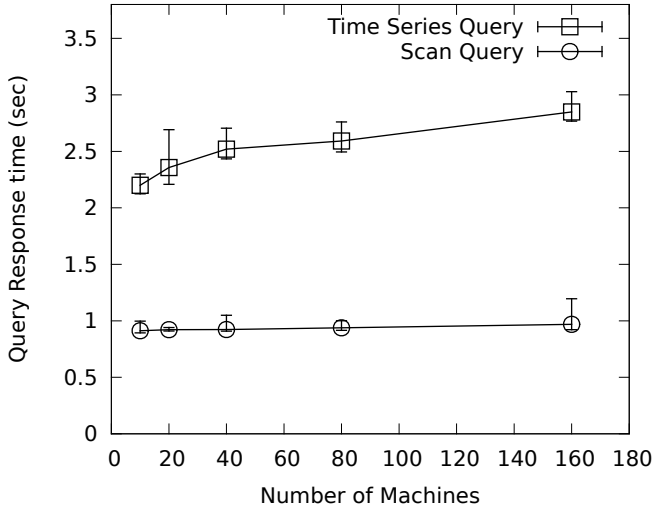

Figure 11: Measuring scale up as the number of machines increases.

3, 6, or 29 hours with 1 minute granularity, graphing one line for each of the top 10 services. The same query is much easier to express in the Scuba GUI, as shown in Figure 9. This query produces 2 aggregation metrics per service, per minute. The limit of $1800 = 180$ minutes $*$ 10 services. This query passes and aggregates 1800 points for each of 2 metrics at every level of the aggregation tree; its aggregation time T_A is noticeable.

5.3 Single client experiments

The first set of experiments test query latency for a single client. For these experiments, we use the 29 hour version of each query and run each query 200 times. We plot the mean response time and the error bars indicate the minimum and maximum response times.

5.3.1 Speedup

We first measure speed up of a single query over data distributed in a 20 machine cluster. We varied the amount of data from 1 GB to 8 GB per Leaf. The total amount of data thus varied from 160 GB to the full 1.2 TB.

Figure 10 shows the results. The time to scan data at each leaf is proportional to the amount of data. The aggregation cost, however, is independent of the amount of data at each leaf; it is a function of the query and the cluster size. In this experiment, the cluster size is constant. With 20 machines and a fanout of 5, there are 3 levels in the tree (1 Root Aggregator, 5 Intermediate Aggregators, and 20 Leaf Aggregators). The scan query passes only one point up the aggregation tree so aggregation takes negligible time. The time series query needs to aggregate a large number of points at every level of the tree, so it takes longer.

5.3.2 Scaleup

We then measure scale up as we vary the number of machines in the cluster from 10 to 160 (doubling the number of machines each time). Each leaf has 1 GB of data.

Figure 11 shows that the time to scan the data (done in parallel on each Leaf) is constant. The aggregation cost grows logarithmically with N. Since the aggregation cost is negligible for the scan query, its response time is constant as the number of machines increases. The time series query, however, needs to aggregate many points at every Aggregator and its response time increases with the number of Aggregators and the number of levels in the aggregation tree. The model presented in Section 4 was very useful in helping us understand these scale up results.

5.4 Multi-Client Experiments

The final experiment tests query latency and throughput as the number of clients increases from 1 to 32. Each client issues 200 consecutive queries with no time between them. For this experiment, we use 160 machines with 1 GB of data at each Leaf. We also use the 3 hour, 6 hour, and 29 hour variants of both the scan and time series queries. The 3 hour variant only needs to scan about 10% of the data at each leaf.

Figure 12 shows the throughput of each query as we vary the number of clients. The first thing to notice is that for each query, the throughput rises as the number of clients increases, until the CPUs at the leaves are saturated. After that, the throughput flattens out. For all queries, the throughput is flat after 8 clients. The next no-

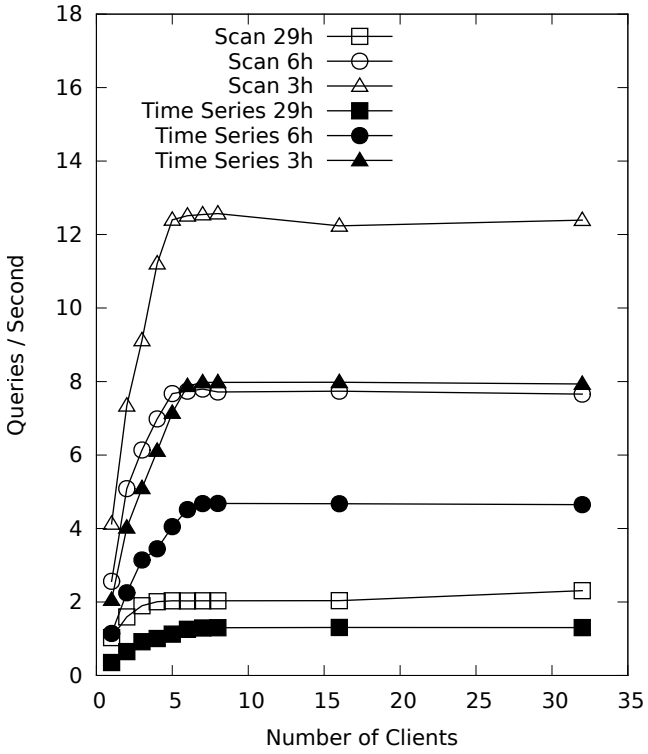

Figure 12: Measuring throughput as the number of clients increases.

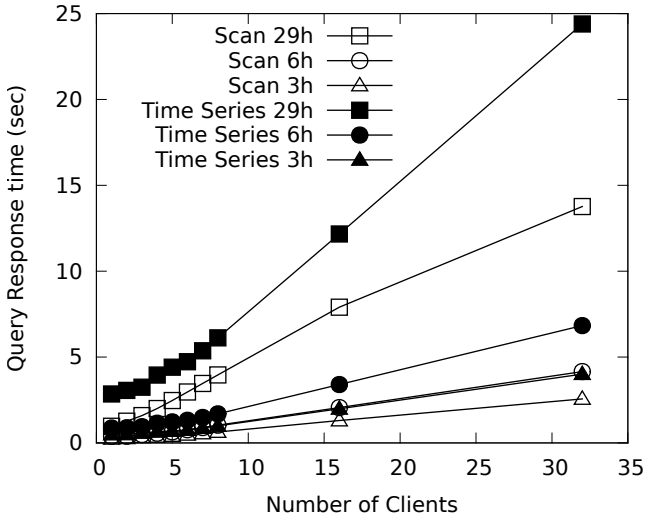

Figure 13: Measuring latency as the number of clients increases.

table point is that the scan queries each achieve higher throughput than their time series query counterparts for the same time range. This point is not surprising, since the scan queries are faster. Finally, for a given number of clients, the throughput for each query type decreases as the time range of the query (and hence the amount of data scanned) increases.

Figure 13 shows that the response time for the queries increases in proportion to the number of clients, as expected.

6. RELATED WORK

The work related to Scuba falls into four categories. First, there are other systems intended for ad hoc, real-time analysis. HyPer [11] also stores data in memory, but on a single, large, expensive machine. Scuba, on the other hand, uses a cluster of relatively cheap, commodity computers and scales easily by adding more machines. HyPer also does not use compression so it requires much more memory for the same amount of data.

Shark [9, 22] and Impala [1] are both intended for real-time analysis over data in Hive. Shark emphasizes completeness of query results and failure recovery and both systems cache data in memory during query processing. However, they both suffer from the long latency to import data into Hive.

Powerdrill [10] and Dremel [12] are two of Google's data management systems meant for analytics. Both are highly distributed, like Scuba, and scale well. Dremel provides a much more complex semi-structured and sparse data model than Scuba; Powerdrill is reported to be much faster than Dremel but still takes about 30-40 seconds/query, according to the paper. In both system, the primary copy of the data lives on disk, so the amount of data stored is less important.

Druid [2] and rrdtool/MRTG (Multi-Router Traffic Grapher) [4, 3] import data quickly, aggregate it on import, and then provide very fast query response time. Druid runs on a fault-tolerant distributed cluster of machines while rrdtool runs on a single machine. Neither can provide drill-downs to the original, raw, data, however, or data types other than strings and aggregated numbers. Scuba requires both for the common use case of tracking changes in performance all the way through stack traces down to the actual code change.

Splunk [6] is also a system for importing logged data, analyzing it, and viewing in graphs and charts. It is intended for data both generated and analyzed in a "cloud." We have no numbers on how fast it either imports or queries the data.

None of the above systems report a way to expire data automatically, unfortunately, which is a key requirement for Scuba, as it is memory-bound and most queries only need data that is 1-2 weeks (or hours!) old.

Second, there are multiple systems that use compression to reduce memory and/or disk footprints. Westman et al [21] used a row-based layout and found that dictionaries provide a good tradeoff between compression ratio and CPU usage. C-Store [18] and Vertica, SAP Hana [15], Dremel, and Powerdrill all take a column-based approach, which we would like to try next to get even better compression.

Third, none of the above systems can trade accuracy for response time, which Scuba does intentionally. BlinkDB [9], however, can produce results over a sampled set of data in bounded time or with bounded accuracy. While BlinkDB needs to precompute stratified samples before it can answer queries, we would like to experiment with its techniques to bound and report Scuba's inaccuracy better.

Finally, all of the data in Scuba is timestamped and many of the analyses are time-based. In the late 1980s, Snodgrass created TQuel [16, 17] to reason about time and intervals. We ought to revisit this work to see if there are features Scuba should incorporate.

7. CONCLUSION

There are multiple ways in which Scuba differs from most database sytems, all of which make Scuba suit our use cases at Facebook.

- Scuba prunes data as fast as it ingests data, since all tables

are stored in memory and memory is the scarce resource. Pruning is automatic, although the parameters for how much data to keep are adjustable per table.

- Scuba expects that many tables will contain sampled data, because storing every event would be too much data. The sample_rate column is treated specially and query results contain both a raw count and a count adjusted for the sample_rates present in the table. (There may be multiple different sample_rates, as many as one per row.)

- Data import is as simple as inserting a logging call for events in code and creating a process to listen for those events. There is no schema declaration needed; the schema is inferred from the logs and it can evolve over time.

- Similarly, a table can contain rows with different schemas, usually because it contains a few different types of events. For example, a table about user requests might have a mobile_brand column that is only populated for requests from mobile devices.

- Visualization of the results is as important as generating them. Scuba currently has about a dozen different ways to visualize data: time series graphs, pie charts, bar charts, flow diagrams, maps, etc. Most interactive queries are asked via the Scuba GUI, not the SQL interface. (The SQL interface is used in scripts.)

- Comparison queries are a first class citizen in the GUI: the GUI can specify and run two queries that differ only in their time range (to show, for example, week over week changes) or in a condition value (to compare, say, users from 2 different countries). The results of both queries are plotted on the same time series graph or displayed with a percentage change in a table.

- Queries are run with best effort availability. If not all leaves are available, then queries return a result over the data that is available, along with statistics about what percentage of the data they processed. Most other approaches, such as attempting to fetch the data from disk on another leaf, would take too long. Scuba's users are generally happy with this approach, which has little effect on aggregates such as averages and percentiles but guarantees a fast response. Future work includes computing and displaying error bars on aggregates derived from partial data.

At the same time, Scuba is not intended to be a complete SQL database. It supports grouping and aggregations but not joins or nested queries. It supports integers and strings but not floats, although it also adds sets and vectors of strings (but not arbitrary nesting of types).

There are changes we would like to make to Scuba. It is currently row-oriented, although we are exploring whether a column-oriented layout might work better. Scuba could use native alerts; right now, users write alerts on top of Scuba query results. Scuba also needs to continue to scale as its user base grows. The model and experiments presented in this paper are a first step in figuring out how well it scales currently.

Nonetheless, Scuba has skyrocketed at Facebook in numbers of users, data, and queries since it was first written two years ago. Scuba provides the flexibility and speed in importing and querying data that is critical for real-time performance and data analysis at Facebook.

8. REFERENCES

[1] Cloudera Impala: Real-time queries in Apache Hadoop, for real. http://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real/.

[2] Druid. https://github.com/metamx/druid/wiki.

[3] MRTG: Multi-router traffic grapher. http://oss.oetiker.ch/mrtg/.

[4] RRDTool. http://oss.oetiker.ch/rrdtool/.

[5] Scribe. https://github.com/facebook/scribe.

[6] Splunk. http://www.splunk.com.

[7] Aditya Agarwal, Mark Slee, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, 2007. http://thrift.apache.org/static/files/thrift-20070401.pdf.

[8] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Communications of the ACM*, 51(12):77–85, 2008.

[9] Cliff Engle, Antonio Lupher, Reynold Xin, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: fast data analysis using coarse-grained distributed memory. In *SIGMOD*, pages 689–692, 2012.

[10] Alexander Hall, Olaf Bachmann, Robert Büssow, Silviu Gănceanu, and Marc Nunkesser. Processing a trillion cells per mouse click. *PVLDB*, 5(11):1436–1446, July 2012.

[11] A. Kemper and T. Neumann. Hyper: A hybrid OLTP-OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[12] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[13] Raghotham Murthy and Rajat Goel. Peregrine: Low-latency queries on hive warehouse data. *XRDS*, 19(1):40–43, September 2012.

[14] Joshua Rosen, Neoklis Polyzotis, Vinayak Borkar, Yingyi Bu, Michael J. Carey, Markus Weimer, Tyson Condie, and Raghu Ramakrishnan. Iterative MapReduce for Large Scale Machine Learning. Technical report, 03 2013. http://arxiv.org/abs/1303.3517.

[15] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient transaction processing in sap hana database: the end of a column store myth. In *SIGMOD*, pages 731–742, 2012.

[16] Richard Snodgrass. The temporal query language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.

[17] Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computing Systems*, 6(2):157–195, May 1988.

[18] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-Store: A Column-Oriented DBMS. In *VLDB*, pages 553–564, 2005.

[19] Jason Taylor. Disaggregation and next-generation systems design, 2013. http://www.opencompute.org/ocp-summit-iv-agenda/#keynote.

[20] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *PVLDB*,

2(2):1626–1629, 2009.

[21] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3):55–67, September 2000.

[22] Reynold Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. Technical report, UC Berkeley, 2012. http://shark.cs.berkeley.edu/presentations/2012-11-26-shark-tech-report.pdf.