Interactive Analysis of Web-Scale Data

Christopher Olston, Edward Bortnikov,
Khaled Elmeleegy, Flavio Junqueira, Benjamin Reed

Yahoo! Research

ABSTRACT

We consider how to support interactive querying over web-scale data. The basic approach is to view querying as a two-phase activity: first supply a query template, and later supply specific instantiations of the template. Interactive responsiveness is offered in the second phase only. While instances of this problem have been studied in the past, e.g., OLAP and web search, we pursue a more general formulation. Our aim is to build a general two-phase query system.

1. INTRODUCTION

Organizations derive significant value from deeply understanding web-scale data sets. A web-scale data set has tables of cardinality 10^{10} or more with perhaps hundreds of columns, populated with web pages, user click events, and other phenomena recorded from the web. Management of data at this scale is achieved via large shared-nothing clusters of commodity computers, often administered as a service by an (internal or external) third party. Current software systems for this context focus on one of two paradigms: real-time processing of simple key-based lookup queries, or offline batch processing of general queries (Figure 1).

For ad-hoc data analysis, real-time processing yields a qualitative benefit compared with batch processing. If the query-response cycle occurs in real time, the user does not need to context-switch while waiting for query results. Instead, she can interact with the data in a continuous fashion: submit a query, view the results, pose a follow-on query, and so on. It is widely acknowledged that interactive analysis sessions constitute the most effective means of understanding a complex data set. Existing interactive data analysis paradigms include on-line analytical processing (OLAP) [3] and dynamic queries in data visualization [16].

Hence the bottom-right quadrant of Figure 1 is the most desirable. Of course it is also the most difficult to achieve—responding to arbitrary queries over web-scale data in real time is infeasible. Fortunately, rather than arbitrary queries, an interactive analysis session usually comprises a series of interrelated queries, which conform—either explicitly or implicitly—to some sort of common *template*. If the template is known in advance, or can be predicted along the way, the system can prepare auxiliary structures (materialized views and indexes) to facilitate real-time query response.

	batch processing	**real-time processing**
lookup queries	any of [2, 4, 5, 11]	BigTable [2] PNUTS [4]
general queries	Map-Reduce [5] Dryad [11]	

Figure 1: Web-scale data processing taxonomy.

OLAP constitutes one instance of this approach. Web search constitutes another instance, where the query template asks to retrieve the top K pages matching a query phrase P, ordered according to a built-in ranking function, with K and P bound at query time. Data visualization with dynamic queries also follows this rubric, but current systems only handle small data sets. Our aim is to enable interactive querying for a much more general class of queries and data than OLAP and web search, and at very large scale.

In terms of handling general queries, the classical *physical database design* problem [1] comes to mind. The standard formulation aims to minimize average query and update cost (where cost may be tied to resource utilization and/or response time) given a model of the anticipated query/update workload, under a space constraint. Our formulation differs in several important ways, the first making the problem much harder, and the rest alleviating the difficulty:

- We impose a hard constraint on query response time, on the order of a few seconds, to ensure interactivity.
- Rather than coping with a mixed query workload, we require the user to declare a single, specific query template in advance. (Targeting a single template enables very aggressive optimizations.)
- The user's query template is subject to negotiation: If the system deems it impossible to achieve interactive response times with the original one, it can ask the user to make modifications. The system may even suggest candidate modifications, i.e., specific filtering, sampling or aggregation steps that improve auxiliary structure size and speed, but retain (some of) the query's usefulness.

This article is published under a Creative Commons License Agreement (http://creativecommons.org/licenses/by/3.0/).
You may copy, distribute, display, and perform the work, make derivative works and make commercial use of the work, but you must attribute the work to the author and CIDR 2009.
4th *Biennial Conference on Innovative Data Systems Research (CIDR)* January 4-7, 2009, Asilomar, California, USA.

- The data is assumed to be static or only updated in occasional large batches, as is typical in large-scale data analysis settings such as warehousing and map-reduce.

In short, our scenario entails *interactive* processing of a *single*, *negotiated* query template over *static* data. The central optimization problem deals with a constraint on response time and maximizes query generality, rather than handling general queries and minimizing average response time.

1.1 Outline

We present an example usage scenario in Section 2. Then in Section 3 we sketch the design of a two-phase query system. We describe techniques for ensuring real-time response in the second (online) phase in Section 4, and give some initial empirical results in Section 5. Lastly we discuss future directions in Section 6.

2. EXAMPLE SCENARIO

Consider the following web-scale data set maintained by a search engine company:

```
pages(url, content, contentType, language, isSpam,
      isDuplicate)
clicks(ipAddress, url, time)
locations(ipPrefix, country)
```

The `pages` table contains one tuple per web page URL, with the raw URL content as well as various extracted features: the content type (text, audio, video, ...); the language used in the content, if known (English, French, etc., or Unknown/Not-Applicable); whether the page has been classified as spam; and whether the page has been classified as a duplicate or near-duplicate of another page. The `clicks` table contains a series of tuples indicating that a user originating at a particular IP address visited a particular URL at a particular time. The `locations` table provides a mapping from IP address prefixes to countries.

The data is kept on a large cluster with thousands of nodes. The software running on the cluster processes ad-hoc queries and scripts submitted by employees.

Suppose a particular employee wishes to explore some characteristics of the web that might influence the design of a future crawler. The characteristics of interest include the pre-extracted features stored with each URL (content type, language, spam tag, duplicate tag), as well several features that need to be computed: number of referring hyperlinks, content of referring anchortext, number of user visits from a given country. She wishes to see which web sites are dominant for a given set of characteristics, and be able to adjust the characteristics interactively and get a rapid response. For example, she may start by looking at dominant web sites referred by French-language URLs, and then drill-down into ones that contain the phrase "telechargement gratuite" ("free downloads") in the referring anchortext. She may spend several hours exploring the data by applying different filters and seeing which web sites surface.

2.1 Underlying Query Template

Figure 2 shows the template that applies to this query session. Cylinders denote tables, $\bowtie$ symbols denote joins, γ symbols denote grouped aggregation, "UDF" symbols denote user-defined functions, and σ symbols denote filters. Each filter is optional, and is governed by predicates supplied by the user dynamically, as part of the data exploration process. A given set of filter predicates yields a particular instantiation of the query template. There are four types of predicates: Boolean predicates for `isSpam` and `isDuplicate`, categorical predicates for `contentType`, `language` and `country`, keyword matching predicates for `anchortext` (i.e., does the anchortext contain a given set of words), and numerical predicates for `inlinkCount` and `clickCount`.

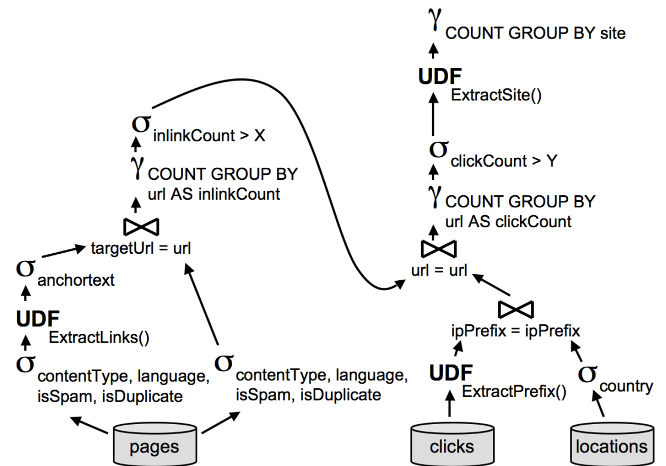


Figure 2: Web exploration query template.

The query template operates as follows. Starting in the lower-left corner, it makes two copies of the web pages table, one to represent referring pages and one to represent target pages. Both copies may be filtered according to optional predicates on content type, language, spam tag and duplicate tag. For the referring pages table, a special UDF `ExtractLinks()` is applied to extract the anchortext and URL of outgoing links. After filtering according to zero or more anchortext keywords, the referring pages table is joined with the target pages table according to the hyperlink reference. Then, the number of pages referring to each page (the *inlink count*) is determined, and pages may be filtered according to a user-supplied lower bound X on inlink count.

Next, moving to the lower-right corner, locations are optionally filtered by country, and then joined with clicks according to IP prefixes extracted from the click IP addresses. The resulting table is joined with the main web page table. Then, the number of clicks to each page (the *click count*) is determined, and pages may be filtered according to a user-supplied lower bound Y on click count. Lastly, a UDF `ExtractSite()` is applied to determine the web site associated with each URL (for example, the web site for `http://www.yahoo.com/games/checkers` is `yahoo.com`), and a final aggregation step determines the number of URLs per site that have survived all the previous filtering steps. The resulting count is the output inspected by the user, who may be interested in all the results or perhaps just the web sites with the highest counts for the given filter instantiations.

3. TWO-PHASE QUERY SYSTEM

In our approach, querying occurs in two phases:

1. **Offline phase.** The user submits a query template. The system then prepares appropriate auxiliary structures, and may in the process negotiate restrictions on the original template.

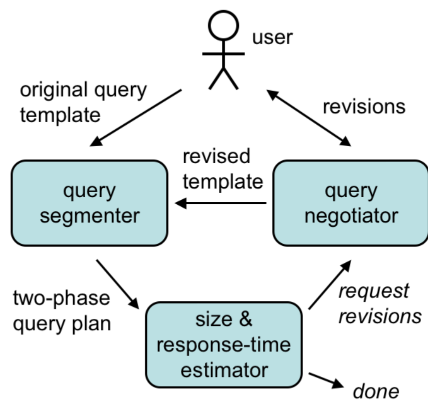


Figure 3: Query planning and negotiation process.

2. **Online phase.** The user poses instances of the query template, and the system uses the auxiliary structures to compile responses in real time.

The offline phase operates over web-scale data in a cluster computing environment. The online phase may also take place in the cluster, provided resource allocation and scheduling techniques that are capable of ensuring interactive response times for all concurrent users. Alternatively, the online phase can take place on a private device operated by the user: the user's desktop, laptop or PDA.

(The primary rationale for *out-of-cluster* online processing has to do with resource utilization: A company with $10,000$ employees has about $10,000$ desktops, and service-oriented or "cloud" computing is unlikely to cause their demise any time soon. A large data-center cluster has 1000–$10,000$ nodes. Utilizing desktops for querying roughly doubles overall processing and storage capacity, or seen from another angle reduces service charges billed from the cluster provider.)

In the out-of-cluster scenario, there is one user and one machine, which leads to both space and time constraints for the online phase of querying. The same can be said of the alternative in-cluster scenario, where there are N concurrent users and M machines, with N and M typically being in the same order of magnitude. In either case, given the limited resources available and the constraint of real-time query response, sufficient work and data reduction needs to occur in the offline phase to allow the online phase to fit within the alloted space and time constraints.

3.1 Query Planning and Negotiation

Figure 3 shows a high-level architecture for query planning and negotiation. When a new query template is received from the user, it first passes through a *query segmenter* module, which produces a two-phase execution plan. A two-phase query execution plan consists of a parameter-free offline portion that terminates in operators which construct a set of auxiliary structures (materialized views and/or indexes), followed by an online portion that reads from the auxiliary structures and produces the final query answer (see Figure 4). Since the real-time response time constraint is likely to be the toughest to achieve, the query segmenter's optimization objective is to minimize work performed in the online phase, with other concerns such as offline costs and offline/online space footprint treated as secondary.

Following query segmentation, a *size and response-time estimator* module determines whether the space and response-time characteristics of the generated plan are acceptable. If not, the query template passes to a *query negotiator* module, which works with the user to restrict the query template so as to reduce its space and time requirements. The query template is then routed back to the query segmenter, and the cycle repeats until an acceptable plan is reached.

3.1.1 Query Segmentation

Our query segmentation problem can be thought of as a simplified version of the classical physical database design problem. In our case, space considerations are ignored and there is only one query template. Hence, rather than an enumerative cost-based search strategy as in [1], a simple rule-based approach may suffice. A basic rubric for transforming a query template into a two-phase plan is as follows (for simplicity we assume the only type of operator whose behavior can depend on parameters is a filter operator, and refer to such operators as *parameterized filters*):

1. Start with a canonical query plan tree.

2. Restructure the plan such that parameterized filters are placed as close to the root (final) operator of the plan tree as possible.

3. Divide the plan into offline and online portions, such that the offline portion incorporates as many operators as possible while remaining parameter-free.

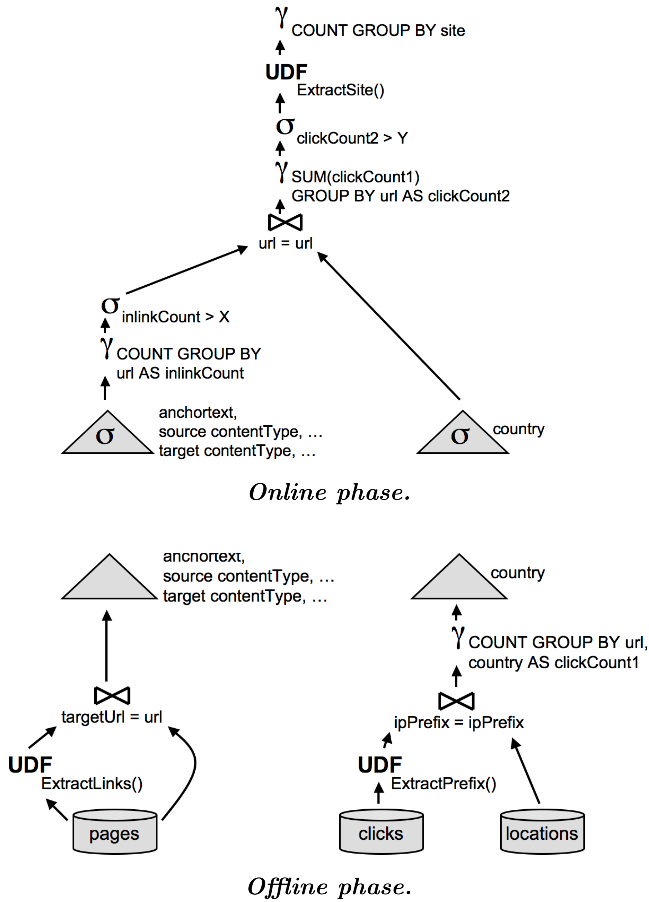
4. Move parameterized filters as close to the leaves of the newly-formed online portion as possible.

5. At each offline/online juncture, introduce indexes (in the case of one or more paremeterized filters) or a materialized view (in other cases).

6. Perform conventional query optimization on the offline and online portions, independently.

The even-numbered steps require some explanation: Step 2 maximizes the size of the offline (parameter-free) portion. Step 4 maximizes the use of indexes for parameter-dependent operations. Step 6 ensures that once the query has been segmented, each segment is maximally efficient.

Figure 4 shows the result of applying this query segmentation heuristic to our example query in Figure 2. Triangles denote indexes, which are constructed in the offline phase and probed in the online phase when specific parameter bindings are supplied. Some items of interest are: (1) partial aggregation of click counts has been pushed into the offline phase, leveraging the *algebraic* [8] property of the COUNT function; (2) although it is not shown in the diagram, early projection introduced in Step 6 can reduce the size of the indexes significantly; (3) if the indexes are clustered on `url`, most of the online phase can employ nonblocking (pipelined) physical operators. Since query segmentation is specialized to a single query template, it can perform highly aggressive optimizations such as items 1 and 3, which a general physical design wizard is unlikely to attempt.



Online phase.

Offline phase.

Figure 4: Two-phase query plan.

3.1.2 Query Negotiation

Query negotiation seeks to reduce the online-phase space and/or time complexity of the user's original query template. One tactic is to suggest the addition of data-reducing operators such as sampling or filtering. For example, in Figure 4 a filter operator that eliminates anchortext terms with low frequency (i.e., occurs in few tuples) can dramatically reduce the size of the index on the left-hand side. Such a modification may have little impact on usability because typically (but not always) only high-frequency terms are of interest to the data analyst. In the same spirit, the index on the right-hand side can be shrunk by eliminating URLs whose click count is below a fixed threshold, since the user may only be interested in URLs with large click count (i.e., large values of Y). Smaller indexes reduce the space footprint and can yield faster query response times. The design of an automatic query negotiator is left as future work.

4. REAL-TIME QUERY ANSWERING

We now describe two techniques to achieve real-time responsiveness in the online phase. The scenario here is to accept query parameters from the user, and derive the answer quickly using indexes that have been constructed in advance in the offline phase. In this section we use the term "query" to refer to the query template instantiation implied by the current parameters supplied by the user, and assume the online execution phase takes place on a single computer.

4.1 Index Ordering to Reduce Seeks

Due to our requirements for extremely fast execution in the online phase, combined with the importance of incorporating unstructured textual data, we are pursuing a solution based on information retrieval (IR) indexes, sometimes called "inverted files." IR indexes incorporate sophisticated compression technology, and are optimized for extremely fast intersection of partial result sets.[1] However, in a traditional information retrieval context the index is only used to retrieve a handful of results (e.g., the top ten search result documents), and consequently IR indexes have not been optimized for retrieving the complete result set.

With an IR index, queries that only match a few indexed tuples (*low-selectivity* queries) are fast. On the other extreme, queries that match a very large number of indexed tuples (*high-selectivity* queries) tend to be fast as well, because they benefit from mostly-sequential disk access and can often be terminated early once a large sample has been acquired (if approximate, statistical answers are acceptable). It is queries with moderate selectivity that can be too slow and make the query interface seem non-interactive, because they must fetch a fair number of tuples, and these tuples are likely to be spread across the disk and require individual seeks. If the query segmenter has not imposed a particular index clustering rule (e.g., cluster by `url` as mentioned in Section 3.1.1), then there is room to choose a physical index layout that alleviates these costly seeks.

For low-dimensional indexing over ordered attributes, special bulk-loading techniques have been devised to reduce the number of random seeks incurred by range queries. For one-dimensional indexes, e.g., B-trees, one simply orders the underlying data by the attribute to be indexed, prior to constructing the index. For indexes with a small number of numerical dimensions, e.g., R-trees [9], space-filling curve techniques such as z-ordering [13] and the Hilbert curve [6] can be used.

In our context, however, because queries define arbitrary subsets of the data, in general there is no way to arrange the data sequentially such that each query's matches are contiguous or even near-contiguous. Fortunately, strict contiguousness is not required—we just need to keep the number of seeks below a threshold. So we need not concern ourselves with low-selectivity queries, and for the remaining queries we can aim to partially cluster the data by query to reduce (but not eliminate) seeks.

In view of the above considerations, we propose the following bulk-loading heuristic, which we call *semi-clustering*.

Let B denote a query parameter binding, represented as a (parameter, value) pair, e.g., (language, Chinese). A query template is instantiated by supplying zero or more bindings (it is fine to leave some parameters unbound; recall from Section 2.1 that each parameterized filter is optional). Let $B(t) \in \{0, 1\}$ be a Boolean variable that indicates whether tuple t matches the filter predicate that B instantiates.

Let s^* denote the maximum selectivity value for which random seeks are sufficiently fast (i.e., a query that incurs s^* seeks still completes within an acceptable response time threshold; a typical value is $s^* = 1000$). Lastly, let $\mathcal{B} = (B_1, B_2, \ldots, B_n)$ denote the list of possible query parameter

[1]Of course, IR indexes are geared toward equality-based lookups. To support range lookups, we will need tree-structured indexes, e.g., B-trees, or a way to handle range queries in IR-style indexes, e.g., [7].

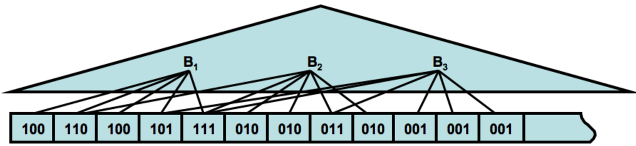


Figure 5: Index with forward bitmaps.

bindings whose selectivity is above s^*, ordered in ascending order of selectivity, i.e., $s^* \leq s(B_1) \leq s(B_2) \leq \cdots \leq s(B_n)$, where $s(B_i)$ denotes B_i's selectivity. For a given tuple t, let $m(t) = \min\{i \in [1, n] : B_i(t)\}$, i.e., the index of the earliest value in $\mathcal{B}$ for which t is a match.

Semi-clustering uses $m(t)$ to determine the order of insertion for bulk-loading the index. (We evaluate the effectiveness of this physical index ordering heuristic in Section 5.1.)

4.2 Adaptive Background Precomputation

Even with the index ordering optimization described in Section 4.1, some queries may be slower than desired. We can take advantage of the fact that users do not pose random queries, but instead tend to pose sequences of related queries. One common pattern is *drill-down*, in which a user starts with an initial query $q_1 = \{b_1, b_2, \ldots, b_k\}$ (where each b_i is a query parameter binding), and then formulates a new query $q_2 = \{b_1, b_2, \ldots, b_k, b_{k+1}\}$, and then $q_3 = \{b_1, b_2, \ldots, b_k, b_{k+1}, b_{k+2}\}$, and so on.

For example, suppose Sue, a member of a search engine company's crawler development team, wishes to understand the crawler's coverage of the China market. She issues the query $q_1 = \{\text{chinese}\}$, and shows the resulting visualization in a meeting. A colleague points out that the presence of duplicate web pages can cause the visualization to be misleading, so Sue refines the query to $q_2 = \{\text{chinese}, \neg \text{duplicate}\}$, causing the display to update accordingly. Another colleague asks whether the visualization includes spam pages, which leads to a third refinement $q_3 = \{\text{chinese}, \neg \text{duplicate}, \neg \text{spam}\}$.

In general, given a sequence of queries $q_1, q_2, \ldots, q_m$ seen so far, the next query is more likely to be a drill-down of one of $q_1, q_2, \ldots, q_m$ than a random query.[2] Given that observation, the system can take advantage of the user's "think time" between queries to compute answers to anticipated drill-down queries. This precomputation work must occur in the background in an interruptible process that is terminated as soon as the user issues a new query.

Since the background precomputation will have limited time to execute, to maximize its effectiveness the system should only precompute queries that are too slow to execute on the fly, i.e., queries with selectivity greater than s^* as defined in Section 4.1 (we refer to these as *dense queries*). Query selectivity can be estimated using various statistical techniques, e.g., [12, 15]. For simplicity, in our implementation we use the hashed counter technique of [14] to identify most (but not all) pairs of attributes that cause query selectivity to fall below s^*, in a time- and space-efficient manner in an initial preprocessing phase.

Given a set of d potentially-dense drill-downs of query q to compute, the obvious strategy is to perform d independent index lookups. We introduce an alternative strategy that enables all d drill-down queries to be computed simultaneously in a single index lookup, by leveraging the fact that the tuples that match a drill-down of q are a subset of the ones that match q. The idea is to store a *forward bitmap* with each tuple t as shown in Figure 5, giving the value of $B_i(t)$ for each $i \in [s^*, n]$.[3] To compute answers to a set of d drill-downs from query q, we initialize d query processors, and use the index to scan the set of tuples matching q. For each match t, we use the forward bitmap to determine the set of drill-downs for which t is a match, and feed t to the corresponding processors.

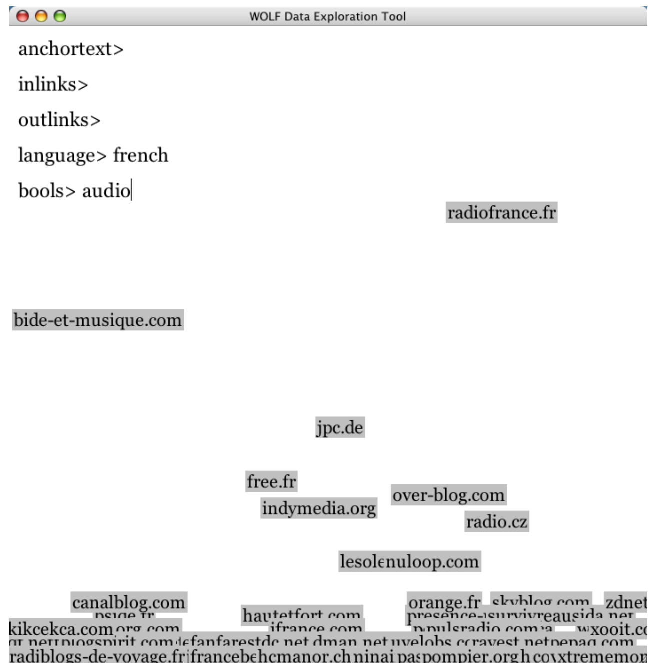


Figure 6: Screenshot of our initial prototype, showing top web sites for French-language audio files. The web sites are arranged in the vertical dimension according to the number of French audio files they contain. The query parameter bindings can be changed dynamically by typing new values after the prompts in the top portion of the screen.

5. INITIAL RESULTS

We have implemented a simple initial prototype; a screenshot is shown in Figure 6. The prototype incorporates the optimizations described in Section 4 for interactive-speed query response, but other functions such as query segmentation are currently done by hand. We give an initial evaluation of the index ordering and background computation techniques (Sections 4.1 and 4.2 respectively), over a 9 GB web crawl data set, which has been derived from a much larger full web crawl data set resident on a 1000-node cluster. The derivation process included UDF processing to extract anchortext strings, projection to eliminate irrelevant columns, and row-wise sampling to achieve further data reduction. The reduced, 9 GB data set is stored and pro-

[2] A similar situation occurs in OLAP, text search and faceted search environments, which is why those systems provide explicit shortcuts for drill-down (a.k.a. "query suggestions").

[3] The bitmaps are typically small, because a small minority of binding values have high selectivity.

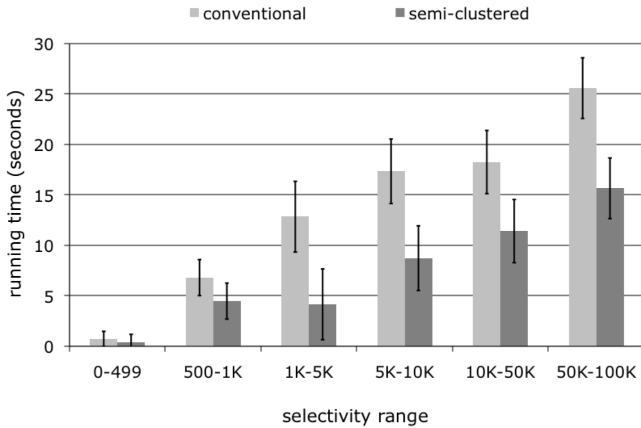


Figure 7: Query performance using standard index (light bars) versus semi-clustered index (dark bars).

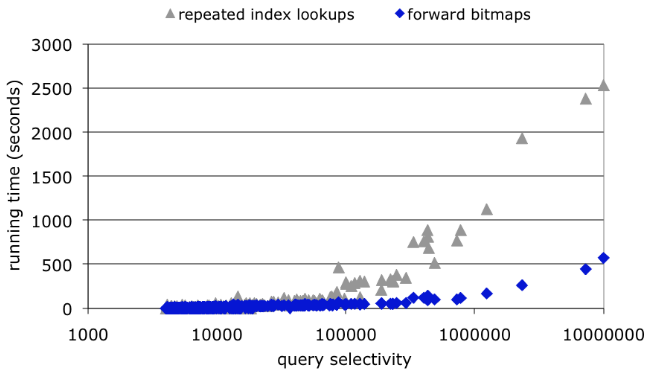


Figure 8: Background precomputation time, using repeated index lookups vs. forward bitmaps.

cessed on a modest MacBook Pro laptop computer, with a 2.16 GHz two-core processor, and 1 GB of physical memory. Our implementation is in Java, and our measurements use a Java Virtual Machine with a 500 MB memory size.

5.1 Index Ordering Effectiveness

Figure 7 shows average query running time for a simplified version of the query template shown in Figure 2, under random vs. semi-clustered index ordering. The x-axis plots selectivity ranges, and the y-axis shows average running time (with confidence intervals). By design, semi-clustering reduces query times most substantially for queries of selectivity just above s^*; in our system $s^* \approx 1000$. In terms of real-time responsiveness, queries of selectivity below 5000 will feel interactive. (Ideally the latency of higher-selectivity queries can be hidden by background precomputation.)

5.2 Background Precomputation Speed

Figure 8 shows the background computation running time under the two approaches using a semi-clustered index, as a function of query selectivity (log scale). Clearly, the forward bitmaps approach permits the set of potentially-dense drill-downs to be computed in a much shorter amount of time, compared to independent index lookups. That said, one advantage of the repeated lookup approach is that if the background computation is halted mid-way, due to the user posing a new query, then at least some of the drill-downs will have been computed.

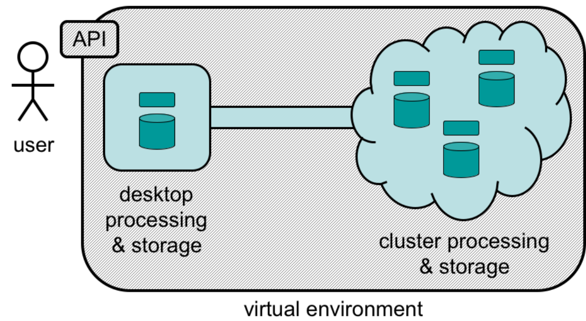


Figure 9: Desktop-cluster virtual environment.

6. LOOKING AHEAD

There are quite a few open questions in the context of this work, some of which we have alluded to already:

- Theoretically speaking, what is the precise boundary between query templates that are amenable to interactivity in the online phase, and ones that are not?

- On the practical side, can an automated query negotiator be built that offers useful suggestions?

- Can we help the user reason about the correctness of a query template before committing to it, e.g., by executing it on sample data with sample parameter bindings?

- Can predictive models be employed to anticipate likely future queries, for which answers (or supporting structures) can be prepared in the background?

- How can we incorporate approximate query processing techniques, e.g., *online aggregation* [10]?

To study these questions and test the overall viability of our approach, we are building a prototype two-phase query system called *Giraffe*. Our initial focus is on out-of-cluster online processing, which mimics the way ad-hoc web-scale data processing is typically done in practice: users migrate intermediate data from the cluster to the desktop as soon as it has been sufficiently reduced by aggregation, sampling and filtering to fit on the desktop, and perform the final analysis steps locally.

Unfortunately, in present practice the user must keep track of the data and processing state in both environments (desktop and cluster), and manually transfer components back and forth. Our system will serve as a single, virtual environment that spans the cluster and desktop (Figure 9). It presents a unified abstraction to the user, automatically divides processing into cluster-side and desktop-side components, tracks status and errors, and automatically stages intermediate data to the desktop for online processing.

Acknowledgments

We thank Brian Cooper for helpful discussions.

7. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated selection of materialized views and indexes for SQL databases. In *Proc. VLDB*, 2000.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. OSDI*, 2006.

[3] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1), 1997.

[4] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *Proc. VLDB*, 2008.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*, 2004.

[6] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proc. PODS*, 1989.

[7] M. Fontoura, R. Lempel, R. Qi, and J. Zien. Inverted index support for numeric search. *Internet Mathematics*, 3(2), 2006.

[8] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1), 1997.

[9] A. Guttman. R-Trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD*, 1984.

[10] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proc. ACM SIGMOD*, 1997.

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. EuroSys*, 2007.

[12] R. Jin, L. Glimcher, C. Jermaine, and G. Agrawal. New sampling-based estimators for OLAP queries. In *Proc. ICDE*, 2006.

[13] J. A. Orenstein. Spatial query processing in an object-oriented database system. In *Proc. ACM SIGMOD*, 1986.

[14] J. S. Park, M.-S. Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. In *Proc. ACM SIGMOD*, 1995.

[15] D. Pavlov, H. Mannila, and P. Smyth. Beyond independence: Probabilistic models for query approximation on binary transaction data. *Knowledge and Data Engineering*, 15(6), 2003.

[16] B. Shneiderman. Dynamic queries for visual information seeking. *IEEE Software*, 11(6), 1994.