LazyBase: Freshness vs. performance in information management

Kimberly Keeton, Charles B. Morrey III, Craig A. N. Soules and Alistair Veitch
Hewlett-Packard Laboratories
first.last@hp.com

Abstract

Information management applications exhibit a wide range of query performance and result freshness goals. Some applications, such as web search, require interactive performance, but may safely operate on stale data. Others, such as policy violation detection, require up-to-date results, but can tolerate relaxed performance goals. Furthermore, information processing applications must be able to ingest updates at the scale of an entire organization. In this paper, we present LazyBase, a system that allows users to trade off query performance and result freshness in order to satisfy the full range of users' goals. LazyBase breaks up data ingestion into a pipeline of operations to minimize ingest time and uses models of processing and query performance to execute user queries. Initial results with LazyBase illustrate the feasibility of the pipelined model, highlight a rich space of trade-offs between result freshness and query performance, and often outperform existing solutions in the space.

1. Introduction

Organizations spend billions of dollars annually to manually locate relevant data, in response to electronic discovery requests, employee document searches, IT management operations and other applications. The primary target of these operations is unstructured information – documents stored on desktops; laptops; email, web and file servers; wikis; and SharePoints across the organization. Many approaches to unstructured information management (e.g., [13]) rely on the ability to scan the contents and activity on machines storing documents to extract descriptive metadata, such as content hashes, term vectors, similarity fingerprints, feature vectors, and usage statistics. This metadata is uploaded to a server and combined with the metadata from other information sources. The metadata can later be queried by users (e.g., eDiscovery legal staff).

Such a system must address several challenges. First, it must support updates from all clients across an enterprise – up to hundreds of thousands for a large enterprise. Furthermore, unlike web crawling and data warehousing applications, which decide what data is most relevant and schedule data ingestion as appropriate, an information management system has no control over data creation rates and cannot determine what is important a priori, necessitating high-performance, scalable ingestion. Second, it may be acceptable to have a delay between when metadata is generated and when it's made visible to users for query, but the results of queries must provide a level of consistency that users can reason about. Third, different applications have different metadata "freshness" and query performance goals, which the system must be able to support. For instance, an enterprise search user may want an interactive response to her query, but be willing to accept results that are a day out-of-date. A virus scanning application that detects an outbreak may want an up-to-date response about which machines contain a copy of the virus within an hour, so that those files can be quarantined to limit the spread of the virus. Different users and different queries may have different goals, even if they rely on the same metadata. The system must determine how to best satisfy the goals of its query workload, or how to best relax some of the goals if all can't be satisfied.

Existing solutions, such as data warehouse extraction, transformation and loading (ETL) processes and web search engines, don't satisfy these requirements. ETL processes periodically ingest data from OLTP databases into a historical data warehouse to provide a fixed result freshness and query performance. Web search engines schedule crawling of pages based on their expected size and popularity, leading to an implicit, but possibly incorrect, assumption that less popular pages can be less fresh than popular ones. This limits required ingest scalability by scheduling data ingest, provides an unknown variation in result freshness, and a fixed query performance point.

In this paper, we describe LazyBase, a system that satisfies these requirements by allowing users to specify their query freshness and performance goals, and then processing metadata to meet these goals. To do so, it breaks up the metadata processing into a pipeline of ingestion, transformation, and query stages, which can be parallelized to improve performance and efficiency. By breaking up the processing, LazyBase can independently determine how to schedule each stage for a given set of metadata, thus providing more flexibility than existing monolithic solutions. LazyBase uses models of transformation and query performance to determine how to schedule transformation operations to meet users' freshness and performance goals and to utilize resources efficiently.

We present initial results that demonstrate the feasibility of our pipelined approach and illustrate the potential benefits of parallelism. These results illustrate a rich space of trade-offs between freshness and performance. We also show that LazyBase outperforms existing database and information retrieval systems for ingesting and querying a range of metadata types.

2. Motivation

Information management applications are representative of a broader class of enterprise applications that often use observational and transactional data to drive various decision support applications. These decision support applications have a wide range of different

Application domain	Desired freshness			
	Immediate	5 minutes	30 minutes	1 day
Enterprise information management	infected machine identification	email	file-based policy violations	enterprise search results, e-discovery requests
Retail processing	purchase transactions	product stocking levels	location tracking information, product search	user comment search, complaint tracking
Social networking	chat, tweets	wall posts, photo sharing	news updates, event invites	friend request, friend search
Data center monitoring	real-time monitoring of local problems	detection of correlated problems	online monitoring graphs	closed trouble ticket searches, knowledge base searches
Online e-tailer	purchase transactions, movie queue updates	top 10 purchased items, wish list updates	best seller lists, product catalog updates	recommendations

Table 1. Freshness requirements for application families from a variety of domains.

requirements for result freshness and query performance. Table 1 provides several examples of this trend from a variety of different domains.

These families of applications operate similarly: observational or transactional updates are ingested into the data store, processed, and then made available for read-only queries. By "observational", we mean updates that are created through observing some aspect of the infrastructure (e.g., the creation of a new file on an enterprise client or the temperature in a particular rack of a data center).

New data is ingested or existing data updated in one of several ways. In some cases, updates from a single client may be batched together and ingested in bulk. Updates may even be batched together across multiple related clients in a coordinated fashion. In other cases, updates may trickle in at a finer granularity, say from a group of interactive users. Scalability is also an issue: updates may be provided by all of the clients in an enterprise, or all of the users of an Internet service. Although the update rate from any particular client may be low, the aggregate update rate across multiple clients may be high. A single piece of data may be updated by a single user (e.g., a website user's configuration preferences), or by multiple users (e.g., multiple users reporting ownership of copies of the same file in an information management application).

Different applications may demand different types of answers. Some require "all information from the start of time to a specified point some time after that." This specific point-in-time may be "now," or some time in the past. Other applications, such as virus propagation discovery, news feeds, and Twitter feeds, may desire only the most recent updates. These models represent different aspects of the freshness continuum.

Some applications can usefully employ partial results (e.g., search, recommendations), and successively refine these results as more information becomes available, or abort a query if the user has sufficient information or decides to change the query. Other applications may need a complete set of results (for a given freshness level) before being able to make a decision. Examples of this latter category include product stock levels and discovery of policy violations across an entire user population (e.g., has a private document been inappropriately disclosed?).

These application characteristics suggest several requirements for the underlying data store that supports them. *Freshness* describes the delay between when updates are ingested into the system, and when they are available for query — the "eventual" of eventual consistency. Application programmers need to be able to specify freshness goals in an easy-to-reason-about manner that puts a bound on how out-of-date results are (e.g., "all results as of an hour ago"). The system should supply results that meet this bound. It may also opportunistically provide even fresher results, if it can do so without performance degradation. The granularity at which users can specify freshness requirements will be driven by the configuration of the system (e.g., five minutes vs. 30 minutes vs. 1 day), and should be designed based on the requirements of the application queries that will be posed.

We note that this definition of freshness is stronger than the traditional notion of "eventual consistency". Eventually consistent systems ensure that updates are eventually applied to the system in a consistent fashion, but provide no guarantees as to when those updates will be exposed to users. Thus, users of eventually consistent systems cannot explicitly request a particular freshness and expect the system to provide it.

Finally, the system must be able to ingest uploads from a large number of clients, where each client upload may consist of a batch of observational records. All records in a client upload should be applied atomically.

3. LazyBase

LazyBase is a distributed database that achieves three goals: (1) efficiently ingest individual client uploads, scaling to handle the large number of clients in an enterprise, (2) provide a "self-consistent" view of metadata to queries, ensuring that all metadata from a client upload is made visible simultaneously across all tables, possibly after a delay from when it was uploaded, and (3) meet users' goals for query performance and result freshness, which may require transforming metadata from an upload-optimized form into a query-optimized form.

LazyBase achieves these goals by trading the freshness of metadata results for query performance. New client metadata is tracked as updates to the base metadata, which can be applied lazily through various processing steps. The system breaks up metadata processing into a pipeline of operations, where each pipeline stage can be parallelized to improve performance, and partitioned to leverage data locality. This section describes the details of LazyBase's design, as well as how it schedules transformation to maintain consistency and meet query goals.

3.1 Summary of processing pipeline

Figure 1 illustrates LazyBase's pipelined design. To maximize ingest performance, data from client uploads is streamed, unsorted, to disk, immediately making the data both durable and queryable (albeit at reduced performance). This first stage of ingestion provides excellent client throughput, focusing on ingest scalability over individual update latency.

Unsorted data is then pushed through the three-stage processing pipeline. First, update data is distributed among sorting nodes to create small sorted update files. Second, sorted files are merged into larger update files. Third, larger merged files are merged with the current authoritative base metadata to create a new authority file, which is then indexed. Each file is equivalent to a database table, storing a row-oriented set of typed columns. Queries consult the

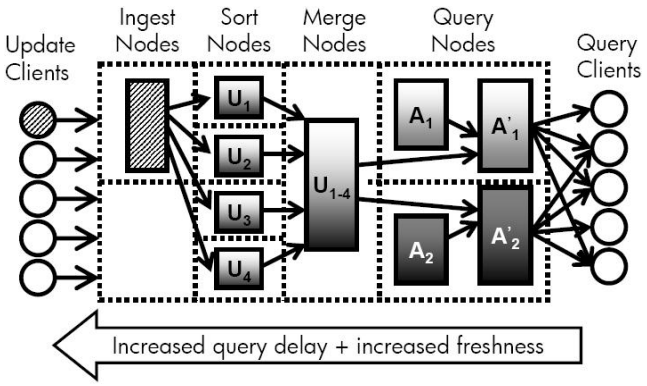

Figure 1. Pipelined architecture of LazyBase. U = update file and A = authority file.

current authority file and optionally the sorted and unsorted update files at each stage for the table(s) being searched.

The remaining sections describe how LazyBase determines which files are used to meet a query's performance and freshness constraints, how it schedules transformation work, and mechanisms it uses for maintaining query consistency through the pipeline.

3.2 Self-consistency

LazyBase's consistency is defined as follows: data from a single client upload is made available in all affected tables concurrently. This approach avoids cases where a table refers to data in another table that has not yet been fully updated. To achieve this goal, the results of a pipeline stage are made available for query only after all data from a single upload has been fully processed at that stage.

3.3 Query processing

Inherent in LazyBase's design is that queries may have to consult several update files to improve query freshness. However, as shown in Section 4, query performance degrades as the number of update files queried increases. Given these trade-offs, LazyBase must answer the following questions: how much consolidation is needed to support the users' desired query performance and metadata freshness constraints? how many (and which) resources should be used to perform this consolidation? These questions must be answered subject to additional constraints such as metadata self-consistency, bounds on file size, available resources, etc. LazyBase treats these questions as optimization problems, where the objective is to maximize the efficiency of performing the consolidation work needed to satisfy users' requirements and other constraints.

To address the first question, LazyBase must look across all queries posed of the system and their requirements to identify a transformation goal for a given set of metadata. Query performance requirements dictate an upper bound on the number of update files that can consulted, while freshness requirements dictate how quickly new data must be transformed to achieve this upper bound. To calculate this upper bound, LazyBase consults a set of *query performance models* (e.g., Figure 2), and then schedules work in the pipeline to meet its goals. Currently, query requirements in LazyBase are known in advance (e.g., the query structure is well defined, such as single term keyword search), and so this work can be lazily scheduled in advance of query execution. Determining a successful schedule for ad hoc queries is an open area of research.

3.4 Pipeline scheduling

Once a transformation goal is determined, LazyBase schedules available processing to achieve the required data layout. LazyBase's worker nodes are stateless, and thus can be assigned to various stages of the processing pipeline when and where needed. Additionally, sorting and merging operations are parallelizable: multiple nodes can be employed to sort multiple files and merging can be done in a hierarchical fashion, where smaller files are merged into successively larger files in multiple passes. LazyBase distributes work based on the size of the data set and the performance of the transform nodes, using *transformation performance models*, which quantify the expected run time of a given operation on a given worker node. Using these performance models, LazyBase assigns when transformation stages should be executed to meet query requirements, as well as how many nodes should be assigned to each transformation. Because computationally similar operations are expected to be repeated frequently, models can be maintained and used in future scheduling.

Co-scheduling updates to a single table together on a fixed set of nodes provides additional data locality benefits by maximizing the use of resources, such as available memory for sorting. Furthermore, it can use intermediate results to reduce network transfer and query node load during merge operations. Examining these and other optimizations is an area of future work.

3.5 Implementation

LazyBase's pipelined architecture requires the ability to break up metadata processing and independently schedule each of the computational stages. Unfortunately, other solutions in neighboring spaces, such as open-source databases and text indexing packages don't easily afford this decomposition opportunity.

Instead, we use DataSeries [1] as a storage layer for maintaining tables. DataSeries is an open-source on-disk format and library for storing and analyzing structured serial data sets. Data sets consist of a set of logical tables, each with a well-defined type composed of named, typed columns. DataSeries' on-disk file format leverages data-specific compression (e.g., relative encoding of numeric data and duplicate string elimination) and generic compression techniques, making it more efficient than other data representations, both in its storage footprint and in the time to read and write data to disk. To trade off between compression efficiency and decompression time for partial data reads, DataSeries works at the unit of extents (often 64KB or more), which are individually compressed. DataSeries files are self-describing, in that they contain one or more XML-specified extent types that describe the extents of the logical tables. DataSeries provides external extent-based indices, which specify the minimum and maximum values within an extent for a given set of columns, stored in a separate DataSeries file. A search on this index provides a direct pointer to all matching extents in the original file, which can then be read directly from disk.

In LazyBase, authority files are maintained as DataSeries files, each corresponding to a single table. Update files are also stored as DataSeries files, organized with the same columns as the corresponding authority file, but with additional timestamp and deletion columns needed to track updates and deletes.

4. Evaluation

LazyBase relies on two sets of models to make its scheduling decisions: query performance models and transformation performance models. In this section, we present measurements of our DataSeries prototype that form the basis of these models. Additionally, we compare our implementation's performance against al-

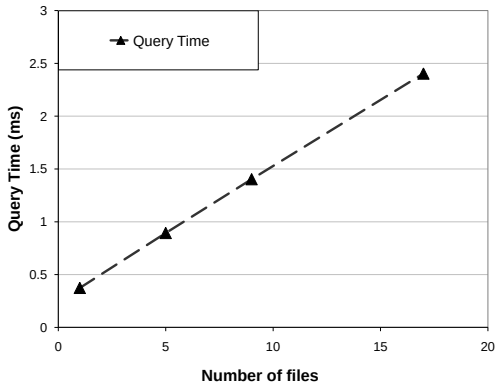

Figure 2. Query performance model for ⟨key, value⟩ data.

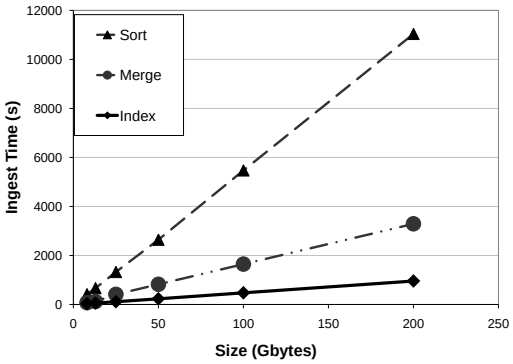

Figure 3. Transformation performance model for sorting, merging, and indexing inverted index data.

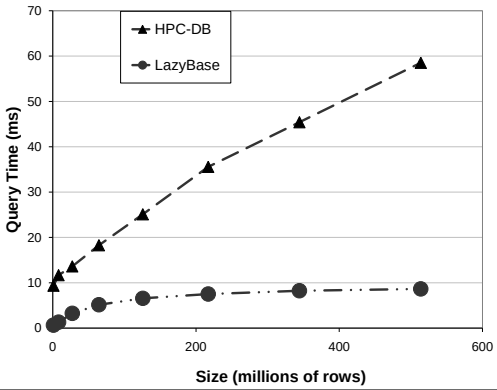

Figure 4. Query performance vs. HPC-DB.

ternative storage layers typically used to store the metadata types of interest.

4.1 Experimental setup

For our measurements, we evaluated three metadata types of interest: traditional ⟨key, value⟩ pairs, tree data (e.g., directory hierarchies), and inverted index data (e.g., mappings of terms to documents containing those terms). For ⟨key, value⟩ pairs, we created a data set containing approximately 500 million 64-bit integer pairs with ever-increasing keys and random values with approximately 15% repeating. For tree data, we created two kinds of trees: deep trees with a fanout of 2 nodes and a depth ranging from 17 to 28, and wide trees with a depth of 4 and a fanout ranging from 100 to 800. Each node in the tree was assigned a random key and value pair from the ⟨key, value⟩ data set. For the inverted index, we created a term/file-position index using a randomly generated corpus of 2.4M text files, with a total of 34 billion total words (about 80,000 distinct words), for a total of 200GB of data.

Queries to these data sets were formulated as follows. We used random single-key queries for the ⟨key, value⟩ data; reported query time is the elapsed time averaged over 1000 such queries issued sequentially through a single client session. A tree query is a single sub-tree query from the root of the tree, which counts the size of the sub-tree. For the inverted index data, reported query time is the elapsed time averaged over seven sequentially issued single-term queries.

All experiments were run on a dual-processor 32-bit 2.4 GHz Dual-Core AMD Opteron machine with 8GB of RAM attached by a Smart Array P800 controller to an MSA70 disk array composed of 16 72GB 15K RPM SAS disk drives using RAID 5 with a 64KB stripe size.

4.2 Performance models

Query performance models quantify query execution time as a function of the number of files that must be consulted to satisfy a query, including the authority file and one or more update files. Figure 2 shows the query performance model for ⟨key, value⟩ data. We see that query performance suffers by just over 1% for each additional file consulted to satisfy the query. The query performance model for inverted index data exhibits similar linear behavior. LazyBase uses these models to determine a transformation goal that meets a user's query performance and result freshness requirements.

Transformation performance models quantify the execution time of sorting, merging and indexing operations for a given metadata type on a given worker node. Figure 3 illustrates the transformation performance models for inverted index data on a single node, which are linear in the size of the data processed. All of the data types we experimented with exhibited similar performance curves. LazyBase uses these models to schedule processing to achieve the data layout required for the query plan determined using the query performance models. This basic single-node performance model can be easily extended to consider parallelism. For example, adding more nodes allows more files to be sorted in parallel. Similarly, a multi-stage hierarchical merge can be modeled by combining the merge costs for each individual stage.

4.3 Comparison with alternatives

We compare the performance of LazyBase to alternatives that typically store the metadata types of interest. For ⟨key, value⟩ pairs and tree data, our comparison point is a high-performance commercial relational database (HPC-DB). For inverted index data, we compare against a Lucene [2] database.

Figure 4 illustrates the comparative query performance of LazyBase to HPC-DB for ⟨key, value⟩ data. Interestingly, LazyBase not only outperforms HPC-DB, but also scales better. LazyBase uses extent-based indexes rather than row-based indexes, resulting in a far smaller index. Although more work must be done to retrieve a single row from disk (it must decompress and scan the extent), the index can fit entirely in memory, resulting in far fewer total disk accesses. LazyBase has lower query cost at small row counts because queries hit in extents that have already been read off disk.

Because LazyBase has direct control over the layout of DataSeries files, it can also specialize its data structures and queries to improve performance. By specializing the tree layout, sub-tree lookup time is 6.5x faster than HPC-DB, across all data set sizes. Similarly, term-based query time is 2x faster than Lucene.

Data type	Improvement with LazyBase
HPC-DB table	3.8x faster
HPC-DB tree	2.5x faster
Lucene	1.3x faster

Table 2. Average data ingest improvement using LazyBase.

Table 2 lists the transformation performance of LazyBase relative to its respective competitor, averaged across all data set sizes. LazyBase improves transformation performance for all of the data sets tested, due to three improvements: type-specific code, larger data pages, and compression. While any or all of these improvements could be applied to the competitor systems, LazyBase's storage layer, DataSeries, provides this functionality by default. DataSeries also stores data in self-contained files, making it ideal for LazyBase's distributed, pipelined design.

5. Related work

LazyBase's use of update tables borrows from similar ideas used in the database and information retrieval communities. The notion of consulting differential files to provide up-to-date query results in large databases with read-mostly access patterns is a long-standing technique [11]. More recent work for write-optimized databases limits queries to the base table, which is lazily updated [9]. Google's BigTable [6] provides a large-scale distributed database that uses similar update and merge techniques, but its focus on OLTP-style updates requires large write-caches and cannot take advantage of the trade-off between freshness and performance inherent in LazyBase's design. Update files are commonly used for search applications in the information retrieval community [5, 10].

These techniques are useful for applications with high update rates that must support user queries, where a limited amount of staleness can be tolerated, or queries can be slowed somewhat. Design choices often dictate a single point along this spectrum – either good query performance with stale results or degraded performance for up-to-date results. LazyBase also trades off query freshness for performance, but it does so in a more flexible fashion, to meet the full range of user requirements.

SCADS [3] describes a scalable, adjustable-consistency key-value store for interactive Web 2.0 application queries. SCADS also trades off performance and freshness, but does so in a greedy manner, relaxing goals as much as possible to save resources for additional queries.

MapReduce [7] and Hadoop [8] provide a data processing mechanism for scalably distributing a transformation (e.g., a sort) across a large number of worker nodes. However, they don't provide a method to coordinate a set of data transformations, as required for self-consistency. Distributed data flow systems, such as Flux [12] and River [4], use a similar event-based parallel processing model for scalability, but are not designed to provide access to the results of intermediate steps, required for the freshness/performance trade-off in LazyBase.

6. Conclusions and open research questions

LazyBase is a metadata management system that trades the freshness of query results for query performance. By breaking up metadata processing into a pipeline of ingestion, transformation and query operations, LazyBase increases its ingest scalability and provides a spectrum of choices in the freshness versus performance spectrum. Users specify their desired levels of query freshness and performance and, leveraging its design and a set of performance models, LazyBase determines how to best meet these goals. Finally, LazyBase's flexible and performant DataSeries storage layer allows it to outperform existing systems for all of the data types examined.

We believe this is the beginning of investigation into a rich space. Many open research questions remain, including: deciding the appropriate processing to meet users' aggregate freshness and query performance goals, choosing the right resource allocation (e.g., degrees of parallelism and partitioning) for metadata processing operations, determining what classes of transformations are amenable to this approach, coping with different priorities of client uploads and user queries, etc.

7. Acknowledgments

We thank Eric Anderson, Michael Armbrust, Armando Fox, Mike Franklin, Mehul Shah, our shepherd Eno Thereska, Joe Tucek, and the anonymous reviewers for their insightful comments on earlier versions of this paper.

References

[1] E. Anderson, M. Arlitt, C. B. Morrey III, and A. Veitch. DataSeries: An efficient, flexible data format for structured serial data. *ACM SIGOPS Operating Systems Review*, 43(1):70–75, January 2008.

[2] Apache Lucene. http:// lucene.apache.org/. 2009.

[3] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. SCADS: Scale-independent storage for social computing applications. In *Proc. CIDR*, January 2009.

[4] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the Fast Case Common. In *Proc. Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, May 1999.

[5] S. Buttcher and C. L. A. Clarke. Indexing time vs. query time: tradeoffs in dynamic information retrieval systems. *Proc. 14th ACM Intl. Conf. on Information and Knowledge Management (CIKM)*, pages 317–318, 2005.

[6] 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*, pages 205–218, November 2006.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*, pages 137–150, 2004.

[8] Hadoop. http:// hadoop.apache.org/. 2009.

[9] S. Hildenbrand. Performance tradeoffs in write-optimized databases. Technical report, Eidgenossiche Technische Hochschule Zurich (ETHZ), 2008.

[10] N. Lester, J. Zobel, and H. E. Williams. In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems. *Proc. 27th Australian Conf. on Computer Science (ACSC)*, pages 15–22, 2004.

[11] D. G. Severance and G. M. Lohman. Differential files: their application to the maintenance of large databases. *ACM Trans. on Database Systems*, 1(3):256–267, 1976.

[12] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. *Proc. ICDE*, pages 25–36, 2003.

[13] C. A. N. Soules, K. Keeton, and C. B. Morrey III. SCAN-Lite: Enterprise-wide analysis on the cheap. *Proc. EuroSys*, pages 117–130, 2009.