Executing Web Application Queries on a Partitioned Database

Neha Narula
MIT CSAIL

Robert Morris
MIT CSAIL

Abstract

Partitioning data over multiple storage servers is an attractive way to increase throughput for web-like workloads. However, there is often no one partitioning that yields good performance for all queries, and it can be challenging for the web developer to determine how best to execute queries over partitioned data.

This paper presents DIXIE, a SQL query planner, optimizer, and executor for databases horizontally partitioned over multiple servers. DIXIE focuses on increasing inter-query parallel speedup by involving as few servers as possible in each query. One way it does this is by supporting tables with multiple copies partitioned on different columns, in order to expand the set of queries that can be satisified from a single server. DIXIE automatically transforms SQL queries to execute over a partitioned database, using a cost model and plan generator that exploit multiple table copies.

We evaluate DIXIE on a database and query stream taken from Wikipedia, partitioned across ten MySQL servers. By adding one copy of a 13 MB table and using DIXIE's query optimizer, we achieve a throughput improvement of 3.2X over a single optimized partitioning of each table and 8.5X over the same data on a single server. On specific queries DIXIE with table copies increases throughput linearly with the number of servers, while the best single-table-copy partitioning achieves little scaling. For a large class of joins, which traditional wisdom suggests requires tables partitioned on the join keys, DIXIE can find higher-performance plans using other partitionings.

1 Introduction

High-traffic web sites are typically built from multiple web servers which store state in a shared database. This architecture places the performance bottleneck at the database. When a single database server's performance is not sufficient, web sites typically partition data tables horizontally over a cluster of servers.

There are two main approaches to executing queries on partitioned databases. For large analytic workloads (OLAP), the approach is to maximize parallelism within each query by spreading the query's work over all servers [11, 17]. In contrast, the typical goal for workloads with many small queries (OLTP) is to choose a partitioning that allows most queries to execute at just a single server; the result is parallelism among a large set of concurrent queries. Queries that do not align well with the partitioning must be sent to all servers. Many systems have addressed the problem of how to choose good partitionings for these workloads [2, 4, 7, 19].

Some workloads, however, execute small queries but do not partition cleanly, as different queries access the same table on different columns. For example, one query in a workload may access a `users` table using the `id` column, while another accesses the table with the `username` column. No single partitioning will allow both queries to be sent to just one server; as a result the workload does not *cleanly partition*. Workloads that cleanly partition allow capacity to scale as servers are added. In contrast, queries that restrict on columns other than the partition column do not scale well, since each such query must be sent to all servers.

This paper suggests the use of *table copies* partitioned on different columns, in order to allow more queries in a workload to partition cleanly. This idea is related to the pre-joined projections of tables used in column stores [20, 23, 29] to increase intra-query parallelism, but here our goal is to increase inter-query parallelism.

In order to exploit partitioned data, including table copies, this paper presents the DIXIE query planner. DIXIE focuses on small queries that can execute on a subset of the servers if the right table copies are available. It addresses an intermediate ground between the whole-table queries of OLAP workloads and the straightforward clean partitioning of some OLTP workloads.

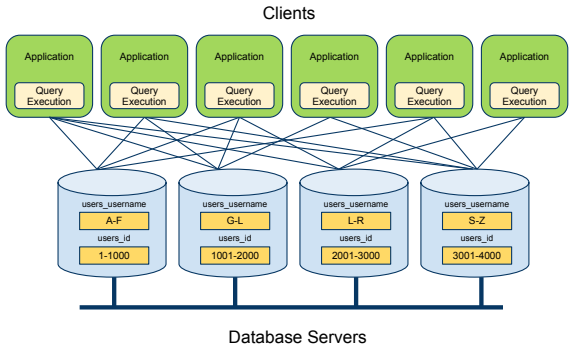

Figure 1: Web site architecture. Front-end web servers running application code issue queries to a cluster of database servers. The users table is copied and partitioned, once by user name and once by ID.

DIXIE uses two key ideas. First, for small queries the overhead at the server may be larger than the data handling cost; these queries do not benefit from distributing the work of an individual query among many servers. As an example, on our experimental MySQL setup, a simple query that retrieves no rows consumes almost as much server CPU as a query that retrieves one row. As a result, cluster throughput can be dominated by the resources wasted due to overhead if small queries are sent to all servers. In the extreme, sending each query to all servers in an N-server cluster may result in $1/N$ the throughput of sending each query to one server.

Second, DIXIE's optimizer uses a novel cost model which appropriately weights query overhead against the costs of data retrieval. For example, DIXIE may prefer a plan that retrieves more rows than another plan, but from fewer servers, if it predicts that the the reduction in overhead from the latter plan outweighs the per-row cost of the former.

We have implemented DIXIE as a layer that runs on each client and intercepts SQL queries; see Figure 1. Applications which use DIXIE can be written as if for one database with a single copy of each table. We have evaluated it on a cluster of ten servers with a traced Wikipedia workload and with synthetic benchmarks. With appropriately chosen table copies, DIXIE provides 3.2X higher throughput for the Wikipedia workload than the best single-table-copy partitioning.

2 Problem

The following example illustrates costs when executing queries over a partitioned database. Consider a simple case with a users table, containing `id`, `group`, `name`, and `address` columns. The `id` column is unique, and the table will be range partitioned over ten servers using some column which we will choose. Assume we would like to issue the following query:

```
Q1: SELECT * FROM users WHERE id = ?
```

Executing many concurrent queries choosing `id` values randomly and uniformly, if we send each query to only one of the ten servers one would certainly expect a throughput higher than if the query was sent to all servers. However, it is unclear exactly how much these costs would differ, since in the ten server case nine of the servers do not need to retrieve or send back any rows.

The cost on the server of executing a simple query like the one above that returns zero data is 90% of the server CPU time of executing a query which returns a small amount of data: 0.36 ms vs. 0.4 ms (these numbers are server processing costs only, they do not include client or network transit time). Section 8 describes the experimental setup. This shows that requesting a row, even if there is no data to read and transfer back to the client, incurs a very significant cost. A profile of the MySQL server shows that the cost consists of optimizing the query, doing a lookup in the btree index, and preparing the response and sending it to the client. On a system executing many concurrent queries, we measure a 9.1X increase in overall throughput if each query is sent to one server instead of ten.

Thus this query would would incur much less cost if the table were range partitioned by `id`, and requests could be sent to one server, as opposed to partitioning on some other column, requiring requests to be sent to all ten servers.

Unfortunately, a single partitioning of a table does not always suffice. Many applications issue queries which access the same table restricting on different columns. Analysis of Wikipedia shows that for a table which comprised 50% of the overall workload, half the queries on that table restricted on one column, half on another. This pattern also occurs in social networking applications with many-to-many relationships [19]. Consider a different query:

```
Q2: SELECT * FROM users WHERE group = ?
```

Partitioning on the `id` column would cause Q2 to go to all servers, while partitioning on the `group` column would cause Q1 to go to all servers. There is no way to cleanly partition this table for both queries.

Storing multiple copies of the `users` table partitioned in different ways can solve this problem. If we create two copies of the `users` table, one partitioned on `id` and the other on `group`, we can efficiently execute both Q1 and Q2. The cost is a doubling of storage space and a doubling in the cost of updates. For workloads dominated by reads the tradeoff may be worthwhile.

With more ways to access a table, query planning becomes more complicated. A smart query optimizer should choose plans which avoid unnecessary lookups, given the appropriate table copies. Properly optimizing these workloads is not just a matter of directing a single query to

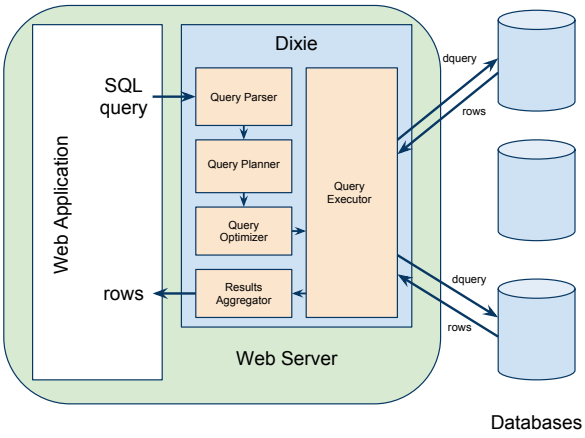

Figure 2: DIXIE's design. There may be many web servers talking to the database server cluster, and thus many instances of DIXIE.

the single appropriate partition. Web applications issue complex queries which often have many potential plans and require accessing multiple servers. For instance, the existence of table copies might affect the table order used in a join, or whether to execute a join by pushing down the query into the servers. The problem DIXIE solves is the selection of plans for executing queries on databases with multiple partitioned copies of tables, with a goal of increasing total multi-client throughput by (among other considerations) decreasing query overhead.

3 Overview and System Model

DIXIE is a query planner intended to operate within a clustered database system. DIXIE takes as input SQL queries written for a single database, plans and executes them on the cluster, and returns rows to the client. Some other mechanism in the overall clustered database handles transactions, if necessary. DIXIE relies on each server in the cluster taking care of local planning and optimization, leaving DIXIE with the task of deciding how to divide the work of each query among the servers.

The key insight behind DIXIE is to reduce the number of servers involved in executing a query by using copies of tables partitioned on different columns, thus improving throughput when there are many concurrent queries. The challenge lies in a choosing a good plan to efficiently use server CPU resources.

Figure 2 shows the architecture of DIXIE. The web application on each front-end web server generates SQL queries intended for a single database server and passes them to the local DIXIE library. DIXIE parses each query, and then the planner generates a set of candidate plans for the query. The query optimizer evaluates the cost of each plan, chooses the plan with the minimum predicted cost, and sends this plan to the executor. The executor follows the plan by sending requests to the cluster of database servers (perhaps in multiple rounds), filtering and aggregating the retrieved rows, and returning the results to the application. Queries generated by the web application are *queries* and the requests generated by DIXIE to the backend database servers are *dqueries*.

The developer provides DIXIE with a *partitioning schema*. This schema identifies the copies of each table and the column and set of ranges by which each copy is partitioned. All copies use range partitioning. DIXIE adds measured selectivity estimates for each column in each table to the schema.

Table copies are referred to by the table name and partitioning key. In the application shown in Figure 1 in Section 1, the `users` table has two copies, one partitioned on `username` and one partitioned on `id`.

DIXIE's goal is to increase total throughput for workloads with many independent concurrent queries, each of which involves only a small fraction on the database. This goal is consistent with the needs of many web applications. Web applications must retrieve data quickly to render HTML pages for a user, and so developers often expect to serve the working set of data from memory. DIXIE's cost estimates assume that most rows are retrieved from memory instead of disk, and that each column used for partitioning has a local index on each server, so that the cost of looking up any row is roughly the same. DIXIE also assumes that applications only issue equality join queries on a small number of tables. These assumptions are consistent with the design of Wikipedia, for example, which does not execute joins on more than three tables, and of Pinax [24], an open source suite of social network applications. Extending DIXIE's cost model to handle workloads which frequently retrieve data on disk is future work. DIXIE handles a select-project-join subset of SQL.

Adding copies of tables can reduce the costs of read queries at the expense of increasing the cost of write queries – writes must be executed on all table copies. Fortunately the applications we examined have very low write rates – by one account Wikipedia's write rate is 8% [6] and based on a snapshot of MySQL global variables provided by the Wikipedia database administrator on 7/11/2011, the write rate was as low as 1.7% (including all `INSERT`, `UPDATE`, and `DELETE` statements). DIXIE is a read query optimizer; but we examine the throughput effect on the server of writing to multiple table copies in Section 8, and show that in a synthetic workload on ten servers with write rates as high as 80% the benefit to reads outweighs the added expense of writing to two copies.

Choosing an appropriate set of table copies and partitions is important for good performance of a partitioned

```
SELECT *
FROM   blogs, comments
WHERE  blogs.author = 'Bob'
AND    comments.user = 'Alice'
AND    blogs.id = comments.object
```

Figure 3: Q3, Alice's comments on Bob's blog posts.

database, but is outside the scope of this work; DIXIE requires that the developer establish a range partitioning beforehand. Our experience shows that potential partitioning keys are columns frequently mentioned in the `WHERE` clause of queries. A subset of DIXIE's techniques would work with another partitioning method like hash partitioning, but in that case DIXIE would not be able to optimize range queries.

4 Query Planning

DIXIE generates a set of plans corresponding to different ways to execute the original query. It estimates the cost of each plan using a cost model, and then executes the lowest cost plan. DIXIE generates plans similar in style to a traditional distributed query optimizer such as R* [17], with a few key differences. First, DIXIE rewrites the application SQL query into many SQL dqueries; essentially it transforms nodes and subtrees in the query plan into SQL statements which can be issued to the RDBMS backend servers. A DIXIE *query plan* is a description of steps to take to execute a query. Second, DIXIE incorporates partitionings of table copies, so it generates plans with different table copies that issue dqueries to subsets of the servers.

As an example, Figure 3 shows a query which retrieves all of Bob's blog posts on which Alice has written a comment. DIXIE will decompose this SQL query into smaller dqueries for each table (or combination of tables) in the query. We assume a partitioned database over N servers with the `blogs` table partitioned on the `author` column and the `comments` table partitioned on the `user` column. One of the plans DIXIE generates for the example query retrieves all of Bob's blogs using the `author` table copy, then retrieves all of Alice's comments on Bob's blogs using the `user` table copy restricting by the blog ids returned in the first step, and finally assembles the results in the client to return to the application. If the tables had copies partitioned on the join keys, it would also generate a plan which sent the join to every server and unioned the results (a *pushdown join*).

DIXIE goes through four stages to generate a set of plans: rewriting the query to separate and group the clauses in the predicate by table, creating different join orders, assigning table copies, and narrowing the set of partitions. The planner generates a *step*, which explains how to access a table, for each table in the query. Each step has a SQL statement to execute on the table, a specific table copy for the table in the step, a list of partitions to which to send the request, a description of what values need to be filled in from previous steps, and what values to save from this step to fill in the next one. A pushdown join step will mention multiple tables.

After DIXIE chooses the generated plan with the lowest estimated cost, its executor saves the results of each step in a temporary table both to fill in the next step and to compute the final result. The query plan specifies which dqueries the executor should send in what order (or in parallel), what data the executor should save, how it should substitute data into the next query, and how to reconstruct the results at the end.

4.1 Query Rewriting

DIXIE seeks to create plans which push down projections and filters into the servers to reduce the amount of data returned to the client. To do this it rewrites each query into queries on each table. If tables have partitionings on join keys, DIXIE will also generate plans that execute the entire join in the database servers (or parts of the join tree in the servers), which we describe in the next section.

Consider a `SELECT` query which uses one table:

```
Q4: SELECT * FROM T WHERE T.a=X
    AND (T.b=Y OR T.c=Z)
```

DIXIE needs to generate a set of dqueries for this query, with each dquery accessing a single table, perhaps on a single partition. To decide what partition(s) a query must use, DIXIE observes that an AND must run on the intersection of the sets of partitions needed by the ANDed expressions, and that an OR must run on the union. To ease this analysis, DIXIE flattens a query's predicate into disjunctive normal form, an OR of ANDs. DIXIE would rearrange Q4's predicate thus:

```
WHERE (T.a=X AND T.b=Y)
   OR (T.a=X AND T.c=Z)
```

DIXIE will create one dquery which retrieves X and Y, and another which can be executed in parallel retrieving X and Z. The results must be unioned together. DIXIE will use the partitioning scheme to determine which partitions should execute each part of the query.

4.2 Join Orderings

DIXIE's current implementation considers all possible join orderings, and creates a plan for each, with each step accessing one table. DIXIE also generates pushdown joins by combining every prefix of the sequence of tables in a join ordering, and creating a plan where the first step of the plan is a pushdown join dquery on the tables in the

```
1  SELECT * FROM blogs WHERE author='Bob'
2  SELECT * FROM comments
    WHERE user='Alice' AND object=?
```

```
1  SELECT * FROM comments
    WHERE user='Alice'
2  SELECT * FROM blogs
    WHERE author='Bob' AND id=?
```

Table 1: Set of plans for the query in Figure 3.

```
1  SELECT * FROM blogs,comments
    WHERE blogs.id=comments.object
    AND author='Bob' AND user='Alice'
```

Table 2: Additional pushdown join plan for the query in Figure 3, with blogs.id and comments.object table copies.

prefix. DIXIE also considers all possible combinations of table copies. If T is the set of tables in the query and there are c_t table copies for table $t \in T$, the original size of the set of plans is:

$$|T| * |T|! * \prod_{t \in T} c_t$$

DIXIE discards any plan with a pushdown join step where there are not matching table copies partitioned on the join keys. For the query in Figure 3 the planner would create the following set of table orderings:

(`blogs`,`comments`),(`comments`,`blogs`)

From that it would generate the two plans shown in Table 1, requesting rows from `blogs` and `comments` in different orders. The pushdown join plan is invalid with the current set of table copies, `blogs.author` and `comments.user`, so DIXIE would prune this plan.

If we had `blogs.id` and `comments.object` table copies, DIXIE would generate sixteen plans, using all combinations of the new table copies, join orderings, and prefixes. In particular, it would generate the pushdown join plan shown in Table 2.

4.3 Assigning Partitions

Every plan is a sequence of steps, one for each table or combination of tables. DIXIE converts this into a sequence of *execution steps*. An execution step is a SQL query, a table copy for each table in the query, and a set of partitions. The planner can narrow the set of partitions based on the table copies and expressions in each step; for example, in the plan where the planner used a copy of table `blogs` partitioned on `author`, the first step of the first plan in Table 1 could send a dquery only to the partition where `blogs.author = 'Bob'`. The set of partitions for each step might be further narrowed in the executor, depending on values that are retrieved and substituted from previous steps. Each execution step also contains instructions on what column values from the results of the previous dqueries to substitute into this step's dqueries, by storing expressions which refer to another table. For example, step two of the first plan in Table 1 would store an expression indicating that the `comments.object` clause required data from the `blogs.id` values that are retrieved in the first step. DIXIE would then convert this into an `IN` expression. Substitution is done during execution. If a step does not require data from any other step, it can be executed in parallel with other steps.

Table 3 shows a set of three plans that DIXIE would generate for Q3 in Figure 3. This table shows the SQL to be issued in the dquery in each step of each plan in the left column. Plans 1 and 2 have two steps each, Plan 3 has one. None of these plans have steps which can be executed in parallel. The middle column uses B, C, and R to represent the intermediate storage for the results as the steps are executed. The second steps of Plans 1 and 2 use `B.id` and `C.object` as placeholders for values returned in the previous steps, which it substitutes into these dqueries during execution. The right column has four parts for each step, and for the purposes of planning we limit our explanation to the first two: the table copy (or copies) used in the step and the partitions to which to send the dquery in the step. We will discuss n_r and n_s in Section 5 when explaining optimization.

Step one of Plan 1 can be sent to just the partition with Bob's blogs, p_1, and step two can go just to the partition with Alice's comments, p_0, independent of the results returned in step one since we intersect the sets of partitions for ANDs. Plan 3 must execute a dquery for every partition because it is not using the table copies which partition on columns used in the most restrictive clauses. DIXIE generates more plans for this query, but these are the three most likely to have the smallest cost, because they use table copies partitioned on columns mentioned in the query.

All of the queries Wikipedia and Pinax issued are simple enough that DIXIE's query planner can efficiently generate a plan for every combination of order of tables in the join and possible table copy. In the applications we examined no query ever joined across more than three tables and no table had more than three copies. However, the number of plans generated is exponential in the size of the number of tables in the query, and existing pruning techniques could be used to reduce the number of plans considered [21].

5 Cost Model

DIXIE predicts the cost of each generated plan using a cost model designed to estimate server CPU time, and chooses the lowest cost plan for execution. DIXIE models the cost

Query Steps		Table Copy Partitions n_r, n_s
Plan 1:		
`SELECT *`		`author`
`FROM blogs`	→ B	p_1
`WHERE author = 'Bob'`		20, 1
`SELECT *`		
`FROM comments`	→ C	`user`
`WHERE user = 'Alice'`		p_0
`AND object IN (B.id)`		1, 1
Plan 2:		
`SELECT *`		`user`
`FROM comments`	→ C	p_0
`WHERE user = 'Alice'`		10, 1
`SELECT *`		
`FROM blogs`	→ B	`author`
`WHERE author = 'Bob'`		p_1
`AND id IN (C.object)`		1, 1
Plan 3:		
`SELECT *`		`id,`
`FROM blogs,comments`		`object`
`WHERE author = 'Bob'`		$p_0, \ldots, p_N$
`AND user = 'Alice'`	→ R	1, N
`AND blogs.id`		
`=comments.object`		

Table 3: Candidate query plans generated by DIXIE for the query in Figure 3

of a plan by summing the costs of each step in the plan, and estimates the cost of a step by summing the query overhead and the row retrieval costs in that step. Minimizing CPU time, rather than elapsed time, is appropriate to the goal of inter-query parallelism.

$$cost_{step} = cost_r * n_r + cost_s * n_s$$

Query overhead, $cost_s$, is the cost of sending one dquery to one server. $cost_r$ is the cost of data retrieval for one row, which includes reading data from memory and sending it over the network. n_r is the number of rows sent over the network to the client, which we assume is close to the number of rows read in the server since most data retrieval is index lookups. Since all rows are indexed and in memory, row retrieval costs do not include any disk I/O costs. DIXIE's optimizer computes cost per step as the sum of the row retrieval cost per row times the number of rows read in the step and the cost of receiving a query at the server times the number of dqueries sent in the step, and the total query cost as the sum of the cost of its steps. It is irrelevant to cost estimation whether the steps in the plan were executed in parallel or sequentially, since we are interested in minimizing overall server CPU time.

Costs are only used to compare one plan against another, so DIXIE's actual formula assumes $cost_s$ is 1 and scales $cost_r$. Section 8 shows how to determine $cost_r$ and $cost_s$. DIXIE uses table size and selectivity of the expressions in the query to estimate n_r, a proxy for the number of rows that might be read in the server. It uses the cost functions below to estimate n_r, the number of rows retrieved, and n_s, the number of servers queried, for each step $step$.

$$selectivity(s) = \prod_{c \in s} \frac{1}{|dk_c|}$$

$$n_r = table\_size_s * selectivity(step)$$
$$n_s = \left|partitions_{step}\right|$$

To compute selectivity, DIXIE stores the number of rows in each table and dk_c, the number of distinct values in each column. DIXIE could be extended to support histograms of values and dynamically updating selectivity statistics, by periodically querying the tables and rewriting the partitioning plan. We leave this to future work. The selectivity function shown assumes a `WHERE` clause with only ANDs, so it can multiply the selectivity of the different columns mentioned in the query.

Table 3 shows three plans for the query shown in Figure 3. Assume the statistics in the partitioning plan predict that Bob has authored twenty blog posts, Alice has written ten comments, and Alice has commented once on one of Bob's blogs. Then the optimizer would assign costs according to the formula described above, as shown in Table 3. The total costs for Plans 1, 2, and 3 are as follows:

$$cost(\text{Plan 1}) = 21 * cost_r + 2 * cost_s$$
$$cost(\text{Plan 2}) = 11 * cost_r + 2 * cost_s$$
$$cost(\text{Plan 3}) = cost_r + N * cost_s$$

6 Query Executor

The executor takes a query plan as input and sends dqueries for each step in the plan to a set of backend database servers. The executor executes independent steps in parallel, and steps which require data from another step in sequence. DIXIE assumes that dqueries request small enough amounts of data that the executor can temporarily store the results from each step in the client. The executor substitutes results to fill in the next step of the plan with values retrieved from the previous steps' dqueries. For example, in Plan 1 in Table 3, the executor would insert `blog_post.id` values from step one into the `comments.object` clause in step two.

The executor can often reduce the number of dqueries it issues by further narrowing the set of servers required to satisfy a step's request. This means that the cost initially assigned to a step by the optimizer may not be correct. For example, the returned values from the first step of a

join may all be on one partition, meaning the executor will only need to send one dquery for the second step to one server, reducing the query overhead and thus reducing the total cost. The optimizer has no way of knowing this at the time when it chooses a plan for execution, and so there are cases where it will not select the optimal plan.

The executor uses an in-memory database to store the intermediate results and to combine them to return the final result to the client. This produces correct results because DIXIE will always obtain a superset of the results required from a table in the join. As it executes dqueries, the executor populates subtables for every logical table in the dquery (not one per table copy). After completion, it uses the in-memory database to execute the original query on the subtables and return the results to the client.

7 Implementation

We have implemented a prototype of DIXIE in Java. It accepts SQL queries and produces dqueries which it executes on a cluster of MySQL databases. DIXIE expects table copies to be stored as different tables in the MySQL databases. The prototype uses JSQLParser [14] to create an intermediate representation of each SQL query. JSQLParser is incomplete, so we altered JSQLParser to handle `IN`, `INSERT`, `DELETE`, and `UPDATE` queries.

We tested the effectiveness of DIXIE using queries generated by Wikipedia and an open source suite of social web applications called Pinax [24], including profiles, friends, blogs, microblogging, comments, and bookmarks.

We implemented a simple partitioner which recommends partitions by parsing a log of application queries and counting columns and values mentioned in `WHERE` clauses. The simple partitioner then generates a partitioning plan with those columns split on ranges to evenly distribute query traffic to each partition. We found DIXIE to be useful in testing different partitioning schemes without changing application code.

DIXIE keeps static counts of number of rows, partitioning plans, table copies, and distinct key counts for each table, for use by its query optimizer. These are stored in configuration files which are read on start up once and not updated. A mechanism to update these configuration files on the fly as table copies are added and deleted or as table counts change could be implemented by regularly rereading the files and updating in-memory data structures to use the new configuration and statistics.

DIXIE's executor saves intermediate results in a per-thread in-memory database, HSQLDB [13]. DIXIE then executes the original query against this in-memory database. An alternate implementation would have been to construct the response on the fly as results are returned from each partition and each step, but using an in-memory database allowed us to handle a useful subset of SQL without having to write optmized code to iterate over and reconstruct results. In the applications we examined, DIXIE never needed to execute a plan which read a large portion of a table into the client, but in a future version DIXIE will do so by requesting tables in chunks.

DIXIE is designed to address the problem of scaling reads. To simulate the costs added by writes, we execute writes sequentially in the client to each table copy, without any serialization between clients. In our experiments, concurrent writes to the same row could cause table copies to become out of sync. We believe this is acceptable since the purpose of this work is to measure the performance impact of added writes in the database servers. The application developer can use existing mechanisms for distributed transactions to manage writes to table copies; this could change what might have been single partition write transactions into distributed write transactions with DIXIE.

8 Evaluation

This section demonstrates DIXIE's ability to automatically exploit table copies to improve database throughput on a realistic web workload. The improvement increases with the number of servers, and is a factor of three compared to the best single-table-copy performance on ten servers and a factor of 8.5 over a single server. This section also explores the factors that DIXIE weighs when choosing among query plans and examines the accuracy of its cost prediction model.

8.1 Workloads and Experimental Setup

Database Workloads. The workload used in Section 8.2 models the Wikipedia web site: it uses a subset of a Wikipedia English database dump from 2008 [1] partitioned across 10 servers (the original Wikipedia database is not partitioned), a trace of HTTP Wikipedia requests [27], and a simulator which generates SQL queries from those requests [6]. The simulator uses the Wikipedia data, published statistics about caching in the application layers, and information from the Wikipedia database administrator to generate an accurate workload. The total database including indexes is 36 GB in size, but the 2008 workload only ends up using a subset of the data that fits in memory. We verify on read workloads that the servers are not using the disk. There are 100K rows in the `page` table, 1.5M rows in the `text` table, and 1.5M rows in the `revision` table (the most heavily queried tables). A majority of Wikipedia queries use the `page` table, restricting on the `page.title` or `page.id` columns. Figure 4 shows the schema of the `page` and `revision` tables.

`INSERT`, `UPDATE`, and `DELETE` queries are 5% of the overall workload, consistent with information provided by

```
page (id, namespace, title, ...)
revision (id, page, text_id, ...)
```

Figure 4: Partial Wikipedia page and revision table schemas.

the current Wikipedia database administrator. Writes are not transactional: the client sends independent writes to each server that needs to be updated. For example, when writing a table with multiple copies, the client sends a separate write to update each of the copies. The rest of the evaluation uses synthetic queries and data constructed to explore specific questions. Every column is indexed.

Hardware. All experiments run on Amazon EC2. Each database server is a "small instance" (one CPU core and 1.7GB of RAM). Experiments use ten servers unless indicated otherwise. Three "large" instances (each with five CPU cores and 12GB of RAM) generate client requests. The clients are fast enough that throughput is limited by the database servers in all experiments. All machines are in the same availability zone, so they are geographically close to each other. Each database server is running MySQL 5.1.44 on GNU/Linux, and all data is stored using the InnoDB storage engine. MySQL is set up with a 50MB query cache, 8 threads, and a 700 MB InnoDB buffer pool. This is sufficient for the working set of Wikipedia data in the workload, because the majority of the database is the text of Wikipedia pages, many of them not accessed.

Runtime Measurement. Before each experiment, DIXIE's planner and optimizer generate traces of plans from application SQL traces. During each experiment, multiple client threads run DIXIE's executor with a plan trace as input; the executor sends dqueries to servers and performs post-processing on the client. We pre-generate plans in order to reduce the client resources needed at experiment time to saturate the database servers. Planning and optimization take an average of 0.17ms per query.

Throughput is measured as the total number of application queries per second completed by all clients for a time period of 300 seconds, beginning 30 seconds after the clients start, and ending 30 seconds before stopping the clients. The traces are long enough that the clients are busy during the duration of measurement. Before measurement, a read-only version of each trace file is run all the way through the system to warm up the database cache and the operating system file cache, so that during the experiment the databases minimally use the disk.

8.2 Wikipedia

Wikipedia's 2008 workload benefits from both partitioning the data over multiple servers and from using more than one table partioning. Figure 5 shows the change in overall throughput for the same workload over 1, 2, 5, and 10

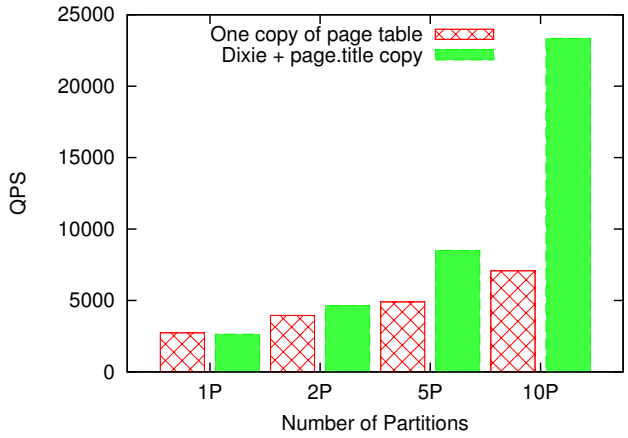

Figure 5: Throughput (queries per second) of the Wikipedia workload with 1, 2, 5, and 10 servers. The patterned red bars correspond to the best setup with only one copy of each table (this setup partitions the page table by page.id). The solid green bars correspond to a setup with an extra copy of the page table, partitioned by page.title.

partitions, with and without an extra table copy. The best partitioning with only one copy per table, which partitions the `page` table by `page.id`, yields a total of 7400 QPS (application queries per second) across ten partitions (see Figure 5, 10P). In this partitioning, 25% of all Wikipedia queries generate dqueries to all partitions. Adding a copy of the `page` table partitioned on `page.title` reduces this number to close to zero, and increases overall throughput to 23347 QPS, a 3.2X increase. DIXIE automatically exploits the table copy to achieve this increase.

In order that the data fit in server memory, the one and two partition cases use a dataset with only 10K rows in the `page` table. With one partition, it is better to have only one copy of each table to avoid extra writes. The benefit of the extra table copy increases with the number of partitions because sending a query to one partition instead of N frees $N-1$ query overhead's worth of CPU time for use by other queries.

To help explain the details of how the added table copy helps, we can divide the read-only portion of the Wikipedia query workload into three parts:

Single table, single-partition queries. These queries access a single table restricting on a column used as a partitioning key. These queries can be sent to one server.

Single table, multi-partition queries. These queries only access one table, but not on any partition key, so they must be sent to all partitions.

Multi-step queries. These are join queries which require accessing two or more tables. They can be executed either as a single pushdown join (which in this workload must be sent to all servers) or in multiple steps. Each step

A	`SELECT * FROM page` `WHERE page.id = i`
B	`SELECT * FROM page` `WHERE page.title = t`

C	`SELECT * FROM page,revision`
	`WHERE page.namespace = 0`
	`AND page.title = t`
	`AND revision.id = page.latest`
	`AND page.id = revision.page`

D	`INSERT INTO page` `VALUES (`c_o`, `c_1`, ... `c_n`)`

Table 4: Examples of Wikipedia page table queries.

is either a pushdown join on a subset of the tables in the join or a set of lookups on a single table, which can be modeled as a single-table query.

We will focus on queries that use the `page` table, which are affected by the addition of the `page.title` table copy. 50% of queries use the `page` table; examples of each type of these queries are shown in Table 4. Queries A and B are single table queries. Query A is a single-partition query, since it can be sent to the single partition containing the appropriate `page.id` value. Query B is originally a multi-partition query. Query C is a multi-step query which is executed as a pushdown join query and must be sent to all servers, and Query D inserts a row into the `page` table.

With the addition of a `page.title` table copy, Query B generates a dquery to one server instead of dqueries to all, reducing query overhead by a factor of ten. Query C requires DIXIE to choose between a pushdown join and joining in the client. In this workload DIXIE always chooses the latter. The next section shows how DIXIE makes that choice depending on the selectivity of the columns mentioned in the query.

Adding table copies can impose a penalty: writes must update all copies. Figure 6 shows the difference in throughput when adding a `page.title` table copy and increasing the percentage of writes. When writes are at 0%, using the `page.title` table copy to direct each query to one partition instead of all partitions achieves a 9.1X improvement in throughput. Throughput improvement decreases as the workload contains a higher percentage of writes. The table shows results from a benchmark with a mix of Queries B and D on the `page` table partitioned over ten servers, randomly and uniformly reading and inserting page rows.

8.3 Query Planning

For some join queries, DIXIE must choose between two plans: a two-step join and a pushdown join. To illustrate this choice, we examine Plans 1 and 3 from Table 3, which DIXIE generates from the application query in Figure 7.

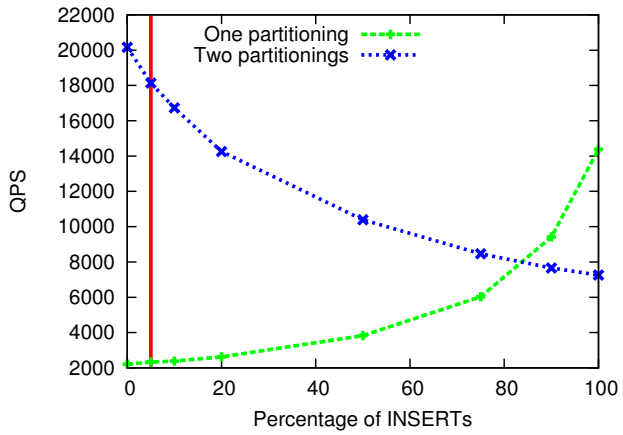

Figure 6: Throughput (queries per second) of a workload executing a combination of queries B and D in Table 4 on ten database servers. One line corresponds to a setup which partitions the page table only on page.id, and the second line corresponds to a setup with an extra copy of the page table, partitioned by page.title. The first setup sends reads to all ten servers and writes to one, the second setup reads from one server and writes to two.

```
SELECT    *
FROM      blogs, comments
WHERE     blogs.author = 'Bob'
AND       comments.user = 'Alice'
AND       blogs.id = comments.object
```

Figure 7: Alice's comments on Bob's blog posts.

If an optimizer were to mainly consider row retrieval cost, it would select Plan 3, the pushdown join, since it retrieves the fewest rows. The plan describes an execution which contacts all servers, but sends at most one or two rows back to the client.

Plan 1 contacts two servers in two steps: first to get Bob's blog posts, then to get Alice's comments on Bob's posts. Plan 1 requires sending back unnecessary rows in step one because it sends back all of Bob's blog posts, even though Alice only commented on one or two.

Figure 8 shows the throughputs for these plans with ten servers, varying the amount of data returned in step one of Plan 1 by increasing the number of blog posts per user. There are 1000 users and ten comments per user. A row in the blog posts table is approximately 900 bytes, and a row in the comments table is approximately 700 bytes. Queries use different values for "Alice" and "Bob" in Figure 7. The graph also shows DIXIE's cost predictions, based on the formula described in Section 5, inverted and scaled up to be in the same range as the measured QPS for comparison.

Figure 8 shows that if the query retrieves few enough rows in step one, the system can achieve a higher through-

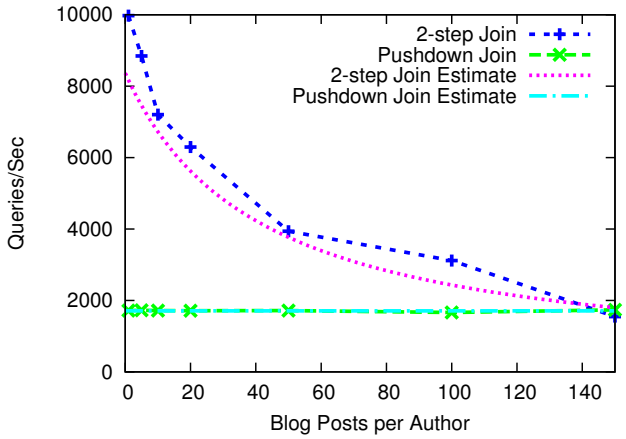

Figure 8: Throughput (queries per second) retrieved from ten database servers and DIXIE's predicted cost for each query plan.

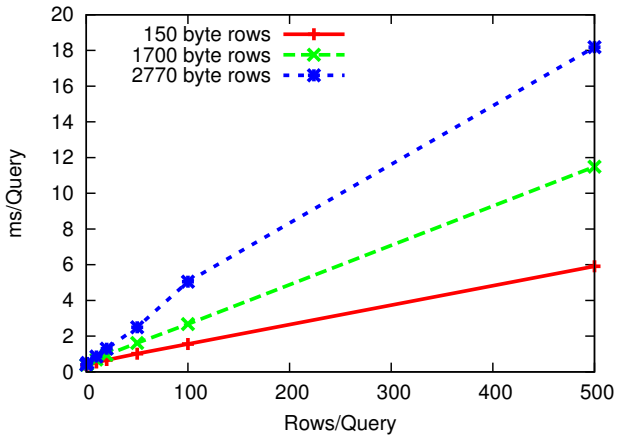

Figure 9: Measurement of milliseconds spent executing a query on a single EC2 MySQL server, varying the number of rows per query and bytes per row using multiple clients.

put with Plan 1 which sends dqueries to only two servers and retrieves more rows, than with Plan 3, which sends dqueries to all servers while retrieving fewer results. Using Plan 1 instead of Plan 3 when there are only ten blog posts per user gives a 4.2x improvement in throughput. This number would increase with more servers. When there are 145 rows returned per author Plan 1 is equivalent in throughput to Plan 3.

Since DIXIE's predicted query costs for the two plans cross shortly after 145 rows, it will usually choose the best plan. Its ability to do so depends on its having reasonable estimates for query overhead and row retrieval time. The next section investigates these two factors.

8.4 Cost Model

To test the accuracy of DIXIE's query cost models (described in Section 5) and measure the cost of per-query overhead, we set up a controlled set of experiments using one database based on the synthetic workloads. We varied the number of rows retrieved, the row size, and the percentage of queries sent to all shards. By varying the number of rows retrieved we can derive a ratio of query overhead to row retrieval time for a single server. We show that query overhead is independent of the size of the rows retrieved.

Setup. The database is on one Amazon EC2 as described in Section 8.1. In each experiment the database has one table of 100K rows, with nine either 15, 150, or 255 character columns. Each column has a different number of distinct keys, and as such a different number of rows returned when querying on that column. Every column has an index and each table fits in memory.

Workload. Throughput is measured by running as many client threads as necessary to saturate the database server (in these experiments 16), each generating and issuing queries of the form:

```
SELECT * FROM table1 WHERE c5 = ?
```

We vary the number of rows retrieved by the queries in a run by changing the restricted column in the query. Within a run each client thread issues a sequence of queries requesting a random existing value from one column (except for the run that measures retrieving zero rows), with a uniform distribution. The overall throughput of a run as measured in queries per millisecond is a sum of each client thread's throughput, measured as a sum of queries issued divided by the number of milliseconds in the run.

Figure 9 shows the time per query measured as $1/qps$ where qps is the throughput in queries per second, as a function of the number of rows returned by the query. This graph shows how total per-query processing time increases as the number of rows retrieved increases for different row sizes. This graph is fit by lines of the form $t_q = t_o + n_r * t_r$, where t_q is the total time of the query, t_o is query overhead, n_r is the number of rows retrieved per query, and t_r is the time to retrieve one row. On our experimental setup, for 150 byte rows, we measure query overhead as 0.45ms and the time to retrieve one row as 0.011ms. For 1700 byte rows, query overhead was .43ms and the time to retrieve a row was .022, and for 2700 byte rows the numbers were 0.5ms and .033ms. DIXIE only uses one value for $cost_r$, but this experiment shows that $cost_r$ varies for row size. Including metrics in the configuration files on the average size of rows in a table and using this in the cost formula would help DIXIE produce better query plans.

Using the formula in Section 5, DIXIE would estimate the cost of retrieving 20 rows from one server as .88, and retrieving 20 rows from two servers as 1.32, a 50% increase. Retrieving 100 rows from two servers instead of

Rows Returned	Servers	DIXIE's Cost	Time
20	1	.88	.95ms
20	2	1.32	1.37ms
100	1	2.64	2.67ms
100	2	3.08	3.24ms

Table 5: DIXIE plan cost estimation vs. actual time.

one server would be 16.7% higher. Table 5 shows the difference between DIXIE's plan costs and experimentally validated plan costs, in milliseconds. The actual time to request 20 rows from one server is .95ms and from two servers is 1.37ms, a 44% increase, and for 100 rows it is a 21% higher.

Query overhead is the MySQL server allocating resources to parse the query, obtain read table locks, and check indices or table metadata to determine if it has rows which match the query.

Since the per-row cost is roughly one twentieth the per-query overhead, DIXIE uses a $cost_r$ of 0.05 in the formula described in Section 5. So in the `blogs` and `comments` query in Figure 7, DIXIE would choose to execute the two-step join plan as long as selectivity statistics indicated that there were less than 162 blog posts per author. Note that for the purposes of showing how costs change according to column selectivity we are ignoring Plan 2, which DIXIE would also consider. This is 12% off from the measured optimal switchover point, shown in Figure 8, which is 145 blog posts per author.

9 Related Work

DIXIE relies on a large body of research describing how to build parallel databases and query optimizers. This section includes the most closely related systems.

DIXIE operates on horizontally partitioned databases on a shared-nothing architecture [22]. The benefits of this design were demonstrated in systems like Gamma [8], Teradata [26], and Tandem [12]. A number of more recent databases exploit horizontal partitioning. H-Store [15], Microsoft's Cloud SQL Server [3], and Google Megastore [10] are main-memory partitioned databases. These systems either prefer or require applications to execute queries which only touch a single partition, and do not describe how to efficiently execute queries that might require spanning partitions.

C-Store [23] and its successor Vertica [29] store copies of table columns partitioned on different columns. C-Store's query planner and optimizer consider which copies of a column to use in answering a query; its focus is parallelizing single queries that examine large amounts of data. Vertica has a sophisticated query optimizer which works over partitioned data on multiple servers, but its optimizer is also designed for large analytic queries, and chooses to favor colocated joins (what we call pushdown joins) where possible [28]. DIXIE relies on the fact that web application queries are often simple, selective, and use a small number of tables in joins. As shown in this work, there are queries where colocated joins are less efficient for these queries than other plans that DIXIE would choose.

The fractured mirrors work [20] presents the idea of storing tables in different formats on different disks to minimize disk seeks, but the authors do not consider partitions, or speak to the costs involved in distributed query execution and how this affects the choice of query plans.

Other work has made the point that social networks do not partition well, and suggested replication solutions [19, 18]. This work relies on network-style clustering in the data, and aims to put users on the same servers as their friends. DIXIE makes no such assumptions.

Schism [7] chooses good partitioning and replication arrangements with the goal of ensuring that transactions need never involve more than one server. Schism doesn't quantify the cost of query overhead or describe how to do query planning and optimization. [2] also investigates partition choice, focusing on warehouse datasets.

In the field of query optimization many systems have addressed how to execute distributed queries. The query optimizer in Orchestra [25], a peer-to-peer database built on a distributed hash table, estimates a plan's cost by considering the cost at the slowest node or link used in each plan stage, which will ultimately optimize for latency but not throughput.

Distributed INGRES [9] has a distributed query optimizer, but like the other systems mentioned above, it optimizes for reduced latency and parallelism. R* [17] is a distributed query optimizer which seeks to minimize total resources consumed, like DIXIE, but does not support the idea of table partitioning, so table access methods are limited. Most of these systems use replication for fault tolerance; none take advantage of table copies on different range partitions to execute plans that minimize machine accesses.

Kossman noted that when estimating costs of a query, communication costs including fixed per-message costs must be considered [16], and discusses choosing which replica of a table to use when executing a query. This paper extends upon that work by noting that in certain web workloads, this cost is the dominant cost of execution, and also proposing a new way of minimizing it.

Evaluation of Bubba [5] showed that when the system is CPU-bottlenecked, declustering degrades performance due to startup and communication costs, which are part of query overhead. DIXIE applies a similar idea to web application workloads, but goes beyond this to motivate keeping many copies of the data, and to use query overhead in the query optimizer to determine cost.

10 Conclusion

Due to the high cost of issuing unnecessary queries in a clustered database, and since web applications often have workloads which do not cleanly partition, developers should use multiple copies of tables partitioned on different columns. DIXIE is a query planner, optimizer, and executor for such a database. DIXIE can execute application SQL queries written for a single database against a partitioned database with multiple partitionings of tables without any additional code by the application developer. DIXIE chooses plans which have high throughput by using per-query server overhead as a dominant factor in calculating query costs. The code is available at `http://pdos.lcs.mit.edu/dixie`.

Acknowledgements

We thank Ramesh Chandra, Sam Madden, and the anonymous reviewers for providing feedback that helped improve this paper. This work was supported by Quanta Computer and by the National Science Foundation.

References

[1] Wikipedia database dump. `http://dumps.wikimedia.org/`, 2008.

[2] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*, 2004.

[3] P. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. Lomet, R. Manne, L. Novik, and T. Talius. Adapting microsoft sql server for cloud computing. In *ICDE*, pages 1255–1263. IEEE, 2011.

[4] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. In *SIGMOD*, pages 128–136. ACM, 1982.

[5] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in Bubba. In *SIGMOD*. ACM, 1988.

[6] C. Curino, E. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In *SIGMOD*, 2011.

[7] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *VLDB*, 2010.

[8] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 1990.

[9] R. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational data base system. In *SIGMOD*, page 180. ACM, 1978.

[10] J. Furman, J. Karlsson, J. Leon, A. Lloyd, S. Newman, and P. Zeyliger. Megastore: A scalable data system for user facing applications. *SIGMOD*, 2008.

[11] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)*, 25(2):73–169, 1993.

[12] T. D. Group. NonStop SQL, a distributed, high-performance, high-reliability implementation of SQL. *Workshop on High Performance Transaction Systems*, 1987.

[13] HSQLDB. HSQLDB. `http://hsqldb.org/`, October 2011.

[14] JSQLParser. JSQLParser. `http://jsqlparser.sourceforge.net/`, October 2011.

[15] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. In *VLDB*, volume 1.

[16] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32(4), 2000.

[17] L. Mackert and G. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB*, 1986.

[18] J. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine (s) that could: Scaling online social networks. *ACM SIGCOMM Computer Communication Review*, 40(4):375–386, 2010.

[19] J. Pujol, G. Siganos, V. Erramilli, and P. Rodriguez. Scaling online social networks without pains. In *NETDB*, 2009.

[20] R. Ramamurthy, D. DeWitt, and Q. Su. A case for fractured mirrors. *VLDB*, 12(2), 2003.

[21] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[22] M. Stonebraker. The case for shared nothing. *Database Engineering Bulletin*.

[23] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented DBMS. In *VLDB*.

[24] J. Tauber. Pinax. `http://www.pinaxproject.com/`, October 2011.

[25] N. Taylor and Z. Ives. Reliable storage and querying for collaborative data sharing systems. In *ICDE*, march 2010.

[26] Teradata. 1012 Data Base Computer, Concepts and Facilities. *Teradata Document C02-0001-05, Teradata Corporation, Los Angles, CA*, 1988.

[27] G. Urdaneta, G. Pierre, and M. Van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11), 2009.

[28] Vertica. `http://www.vertica.com/2010/04/28/vertica-under-the-hood-the-query-optimizer`, April 2010.

[29] Vertica. Vertica. `http://vertica.com`, October 2011.