Adaptively Parallelizing Distributed Range Queries

Ymir Vigfusson
Cornell University
ymir@cs.cornell.edu

Adam Silberstein, Brian F. Cooper, Rodrigo Fonseca
Yahoo! Research
{silberst,cooperb,rfonseca}@yahoo-inc.com

ABSTRACT

We consider the problem of how to best parallelize range queries in a massive scale distributed database. In traditional systems the focus has been on maximizing parallelism, for example by laying out data to achieve the highest throughput. However, in a massive scale database such as our PNUTS system [11] or BigTable [10], maximizing parallelism is not necessarily the best strategy: the system has more than enough servers to saturate a single client by returning results faster than the client can consume them, and when there are multiple concurrent queries, maximizing parallelism for all of them will cause disk contention, reducing everybody's performance. How can we find the right parallelism level for each query in order to achieve high, consistent throughput for all queries? We propose an adaptive approach with two aspects. First, we adaptively determine the ideal parallelism for a single query execution, which is the minimum number of parallel scanning servers needed to satisfy the client, depending on query selectivity, client load, client-server bandwidth, and so on. Second, we adaptively schedule which servers will be assigned to different query executions, to minimize disk contention on servers and ensure that all queries receive good performance. Our scheduler can be tuned based on different policies, such as favoring short versus long queries or high versus low priority queries. An experimental study demonstrates the effectiveness of our techniques in the PNUTS system.

1. INTRODUCTION

Database tasks that are relatively well understood in traditional settings become significantly challenging again at web scale. The need to provide access to terabytes of data for millions of users a day with very low latency requires rethinking many aspects of a database system's architecture and algorithms. This challenge has led companies and researchers to develop new database systems that can scale to the level required by web workloads: Google's BigTable [10], Amazon's Dynamo [13], Microsoft's Azure SDS [4], Facebook's Cassandra [20], our own PNUTS [11], and others. These systems are all shared-nothing databases, so that scalability can be achieved by adding more servers with local-attached disk.

One task that is important in web workloads, but difficult to execute efficiently at scale, is answering range queries over ordered data. Range queries are important in a variety of web workloads; some examples include:

- **Time-ordered data** - We might have a table of items for sale, sorted by listing date, and users query for the most recent items or items that are new since the last time they queried.
- **Secondary indexes** - We can create an ordered table that is an index over a secondary attribute like price or average customer rating. Then, users can ask queries for specific price ranges or for items above a certain rating.
- **Hierarchical clustering** - We can create a table for a Craig's List-style posting site, where keys are a composite of location, description, and item id (e.g., `CA.southbay.sanjose.autos.id654`), and other fields list attributes, such as brand. A user searching for a common brand, such as "Toyota," can scan a small range of the table (i.e., `CA.southbay.sanjose.autos.*`) with a predicate on brand and retrieve a large number of results. A user searching for a rarer brand, such as "Porsche," must scan a larger range (i.e., `CA.*`), to find enough results.

In each case, there could potentially be a large number of results: hundreds of thousands of listings in the last 24 hours, products under $10, or items in CA. Moreover, we might specify a predicate over the range (such as "find products under $10 where category='toys' and quantity>0"). If the predicate is selective, we might have to scan a large range just to find a few results. For these reasons, a range scan might have to scan a significant fraction of the table. However, many of these queries support online applications where there is a user waiting for (at least some) results to appear in a web page. Thus, not only do we want to return some results quickly, but we want to complete the entire range query quickly.

In a massive scale web database such as those mentioned above, a database table is partitioned over many servers. To meet our response time requirements, it makes sense to execute a range query by sending the query in parallel to several servers, and returning results to the user as soon as they return. This approach raises a fundamental question: how parallel should the query be? If we send the query to too many servers, results will be returned faster than the

VLDB '09, August 24-28, 2009, Lyon, France

client (typically an application server) can consume them, wasting server and network resources that could be allocated to other, concurrent queries. If we send the query to too few servers, the client will be waiting for results that trickle in, especially if the predicate is very selective.

We have built PNUTS, a massive scale shared nothing database that is in production serving web traffic for some of Yahoo!'s applications. In this paper, we describe extending PNUTS to provide parallel range queries. Our approach to the question of parallelism is to adaptively tune the parallelism for each query during query execution, depending on the sustainable data transfer rate and number of competing concurrent queries. The first aspect of our solution is **adaptive server allocation**, to find the ideal parallelism for a particular query. Our system stores an ordered table using range partitioning, storing multiple (e.g., a few hundred) partitions per storage server. For a given query q submitted by a client, our range query engine chooses an initial parallelism level (called K_q), and contacts K_q servers in parallel to scan one partition per server. As the scanning proceeds, the range engine adjusts K_q upwards or downwards to match the aggregate rate at which servers are sending data to the rate at which the client can consume data.

Even after we know the ideal parallelism level K_q for a query, we have to be careful about assigning the same server simultaneously to too many queries, as this will lead to disk contention, slowing down all of the queries. We validate experimentally the intuition that limiting the number of concurrent scans on a server to avoid random I/O can result in faster execution for all queries. Thus, the second aspect of our solution is **multi-query scheduling**, to determine which servers to allocate to each query at a given point in time. Our scheduler assigns no more than L_s concurrent queries to a server s, where L_s is a parameter we can set depending on the server's capacity. Then, when a query finishes scanning a partition on a given server, we allocate that server to the same or a different query. The scheduler allows us to tune our preference for different kinds of queries, for example to prefer short versus long (to prioritize interactive queries) or to prefer high priority queries versus best-effort queries. Our scheduler must be adaptive as new queries constantly enter the system.

Although parallel database systems have supported parallel scans for years [3, 5, 8, 9, 14, 15], the focus of previous work has typically been on laying out data to allow maximum parallelism [7, 17, 21, 18]. In a massive scale database such as PNUTS, achieving maximum parallelism is not desirable; there are so many storage servers that scanning them all in parallel can easily overload any client. Moreover, massive scale web databases typically have very many concurrent clients querying the database. For these two reasons, we must throttle the parallelism level to get good performance for everybody, and finding the best way to throttle parallelism is the focus of this paper.

Our approach does not require us to maintain data statistics (such as value histograms) or system statistics (such as the peak bandwidth available to servers.) Although such statistics could be used to improve our adaptivity (and in particular, can help us choose a good initial K_q), accurate statistics are difficult to maintain, especially in a large scale distributed system. Moreover, we cannot rely on statistics alone to choose the optimal parallelism level. In particular, statistics are a blunt instrument that may not correctly predict conditions at query time. For example, the rates at which servers can send or receive data depend on transient conditions such as the number of concurrent range queries, the length of the ranges, the load of concurrent non-range queries (such as single record reads and writes), network cross-traffic, and so on. Similarly, data parameters such as predicate selectivity or the number of tuples in a range depend on data distributions which might change as certain items become more numerous or popular. In this paper, we focus on a "statistics-free" implementation of an adaptive range engine, and demonstrate that it provides good performance. In the future, we can examine whether the benefits of maintaining statistics are worth the cost.

In this paper we describe techniques for adaptively optimizing the execution of range queries in a large scale distributed database. We make the following contributions:

- We present an architecture for adaptive execution of range queries in a massively parallel database.
- We describe an adaptive server allocation algorithm that matches parallel server sending rates with client consumption rates to optimize a given query, and demonstrate its effectiveness experimentally.
- We develop a scheduler and scheduling metric with a tunable parameter to favor certain queries over others. Theory and experiments show that the scheduler effectively optimizes multiple queries.

The rest of this paper is organized as follows. In Section 2, we describe our system and our range query requirements. Then, in Section 3, we present our algorithm for adaptive server allocation. In Section 4, we describe a scheduling algorithm for handling multiple concurrent queries. We present experimental results in Section 5. In Section 6 we examine related work, and we discuss our conclusions in Section 7.

2. SYSTEM AND PROBLEM OVERVIEW

In this section, we discuss the design of PNUTS as it relates to range queries. First, we describe the data and query model supported by the system, and then present an overview of the distributed architecture. Finally, we focus on the execution of range queries in this architecture. A more complete description of PNUTS itself is presented elsewhere [11], and focuses on issues such as fault tolerance, the transaction model, data replication, operational issues, etc. While we discussed range query support in this previous work, we did not examine range query optimization.

2.1 Data and query model

PNUTS manages tabular data, where each table is a collection of records each identified by a unique primary key. The system is designed primarily to support web serving workloads, where the primary operations are reads and writes of individual records and scans of subsets of the table. While we support scans of the entire table (for example, to support MapReduce [12] computations), PNUTS is not optimized for this workload, and does not in general support complex analytical queries. Instead, the system is focused on providing low latency for a few operation types:

- **get** - retrieve a single record by primary key
- **set** - insert or update a single record by primary key
- **delete** - delete a single record by primary key

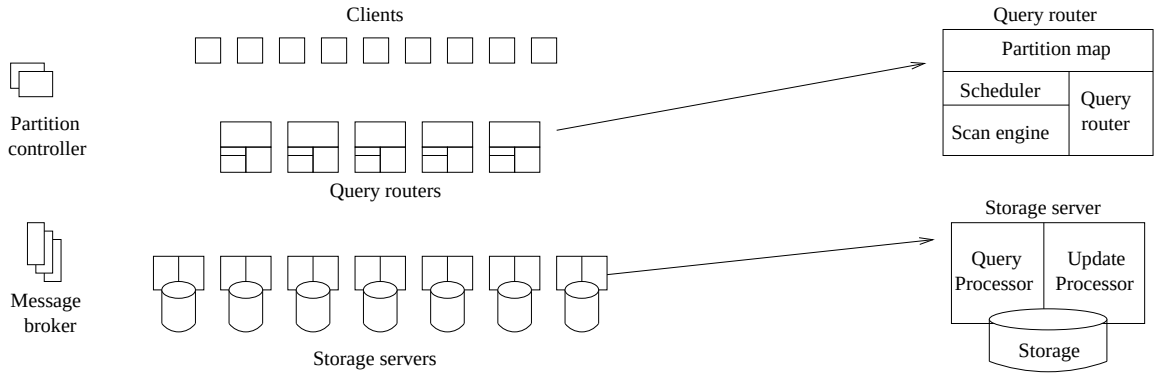

Figure 1: System architecture

- **getrange** - retrieve all records whose primary key falls in the specified interval and which pass an (optional) predicate

Of course, a **getrange** query that scans a non-trivial subset of the table will take some time to return all results. Thus, "low latency" in the context of **getrange** really means that the client should not have to wait too long for results: the first results should return quickly; we should avoid significant wait times for subsequent results; and the whole scan should complete relatively quickly. In order to get first results fast, we do not guarantee that records are returned in key order. It is possible to add additional constraints to our scheduler to ensure ordering or provide a top-K abstraction, although we have not yet implemented this.

Range and table scans do not guarantee a consistent snapshot of the data. PNUTS provides only per-record consistency guarantees, such that updates to a given record are transactional but no ACID-style guarantees are offered across records. This decision improves scalability, as we need not manage locks or transaction state across servers. As a result, a scan may return different records with states representing different points in time, may not return some records inserted after the scan began, and may return some records deleted before the scan ended. While we may add multi-record transactional guarantees in the future, per-record consistency is sufficient for many web applications.

2.2 System architecture

A key goal of PNUTS is elastic scalability, which means that we can add capacity easily by adding more servers. This means that every component on the data path of a request must be horizontally scalable. For the specific case of range scans, we can improve the performance of single and multiple range scans by increasing the number of servers and thus degree of parallelism that can be brought to bear.

Figure 1 shows the architecture of the system. A data table is horizontally partitioned and stored on multiple *storage servers*; a storage server typically holds many partitions (e.g., a few hundred 1 GB partitions). The *partition controller* holds the authoritative mapping of partitions to storage servers (the "partition map"). In our implementation, this mapping is stored using a B+tree in MySQL. However, the partition controller is not part of the data path of requests. Instead, the *query router* accepts requests from clients and uses a cached copy of the partition map to forward single record requests to the storage server which currently holds the appropriate partition. If a query router receives a **getrange** request, it is forwarded to the *scan engine* for processing. The level of indirection provided by the query router lets us move partitions between storage servers for load balancing or recovery without impacting the client's operation. On a storage server, read requests (e.g., **get** or **getrange**) are handled by the query processor, which retrieves data from cache or disk. Write requests (e.g., **set** or **delete**) are handled by the update processor, which transmits the update to the message broker for propagation to other replicas before writing to local storage.

The *scan engine* of the router receives **getrange** requests; uses the partition map to determine which partitions overlap the range and thus which storage servers need to be contacted to answer the query; makes parallel connections to each of the storage servers; collects the results; and returns them to the client. The scan engine uses the *scheduler* to determine which and how many storage servers are to be contacted in parallel. In this paper, we will focus on the operation of the scan engine and the scheduler.

2.2.1 Requirements for supporting clients

The clients of the system are typically application servers, usually with multiple application servers per application. The system is also multi-tenant, supporting multiple applications and workloads on the same set of PNUTS servers. These client characteristics result in several requirements that are relevant to range scans:

- We must cope with many concurrent clients executing both single-record (e.g., **get**) and multi-record (e.g., **getrange**) requests.
- We must deal with a variety of request characteristics. For example, with range scans we must handle short and long ranges, selective and un-selective predicates, and so on.
- We must handle clients with varying capabilities; in particular, some clients can handle a high rate of results while others cannot. This variation results from differences in client hardware, client machines that are more or less overloaded, client machines that do a lot or little processing of each result, etc.

These requirements make it difficult to optimize *a priori* for a particular workload and help motivate our adaptive approach to optimizing range scans. In particular, our adaptive techniques must not only optimize the performance of a

single client or class of clients, but must also provide good, fair service to a variety of clients.

2.3 Executing range queries

A straightforward mechanism for executing range queries is to scan one partition at a time, returning results to the client during or after the scan of each partition. While this mechanism is relatively easy to implement, it does not take advantage of the inherent parallelism of the system to reduce total query execution time. Moreover, if a client specifies a selective predicate, we may have to scan multiple partitions before finding any results, and the client will experience a long response time before the first result.

Instead, our scan engine sends scan requests to multiple storage servers in parallel. We define L_s as the limit on the number of concurrent queries that can be handled by a given storage server s, depending on its disk and processing bandwidth[1]. The flow of a range query is:

1. The query router receives a query q, which is a **getrange** request, and passes q to the scan engine.
2. The scan engine looks up the endpoints of the range of q in the partition map to find the $start_q$ and end_q partitions. The set of partitions $\mathbf{P}_q$ in the key range between $start_q$ and end_q (inclusive) are the partitions that must be scanned for q.
3. The scan engine chooses an initial parallelism level K_q.
4. The scan engine asks the scheduler for K_q partitions to scan from the set $\mathbf{P}_q$. The scheduler returns $\mathbf{SP}_q$, a set of *(storage server, partition)* pairs, using our scheduling algorithm (Section 4). A storage server may appear in multiple pairs in $\mathbf{SP}_q$ if there are multiple partitions to be scanned on the same server. A server s may appear no more than L_s times in the union of **SP** sets currently assigned to all queries. It may be that $|\mathbf{SP}_q| < K_q$ if we cannot satisfy the requested K parallelism level for all queries given the L_s limits. Then, K_q is the *desired parallelism* and $|\mathbf{SP}_q|$ is the *actual parallelism*.
5. The scan engine forwards a **getrange** request to the storage servers in $\mathbf{SP}_q$, specifying for each the partition to be scanned and the predicate to be applied to records.
6. Each storage server begins scanning the specified partition, streaming results that pass the predicate back to the scan engine and on to the client.
7. When a storage server completes scanning a partition the *(server,partition)* pair is removed from $\mathbf{SP}_q$. If this reduces the parallelism of the query below its current desired K_q, the scheduler may choose another partition from $\mathbf{P}_q$ and add the $(server, partition)$ pair to $\mathbf{SP}_q$. If so, a new **getrange** request is sent to this new server, which begins scanning and returning results.
8. Concurrently, we adapt K_q, the desired parallelism level for q, based on the adaptive server allocation algorithm of Section 3. If a new K_q is chosen, the scheduler will try to add or remove $(server, partition)$ pairs to/from $\mathbf{SP}_q$ so that $|\mathbf{SP}_q| = K_q$, again subject to the constraints of other queries and the L_s limits. This new $|\mathbf{SP}_q|$ does not cause any current partition scans to be terminated; instead, the scheduler will try to increase or decrease (as appropriate) the number of range scanning servers as soon as feasible.
9. When all the partitions in the interval $[start_q, end_q]$ have been scanned, the query q terminates.

[1]In practice, we use the same L_s value for similar servers. We evaluate appropriate L_s values in Section 5.

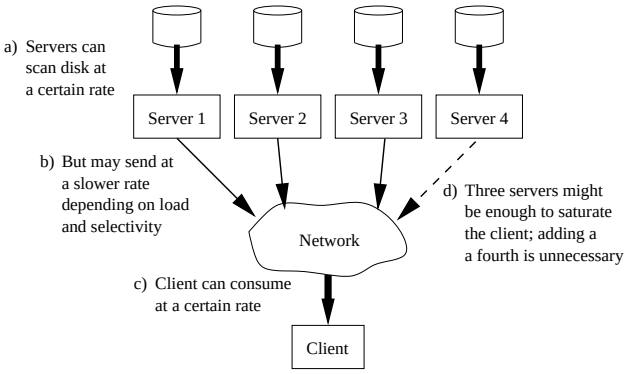

Figure 2: Adaptive server allocation assigns the minimum number of servers needed to saturate the client.

Our adaptive algorithms determine how many and which storage servers to contact in parallel; these algorithms are described in the next sections.

3. ADAPTIVE SERVER ALLOCATION

We now describe the mechanism for determining the ideal parallelism level for a query execution. Note that the same query, executed at different times or by different clients, might result in a different ideal parallelism level. Therefore, during every execution of a given query we adaptively compute how many servers to allocate. However, when there are multiple concurrent range queries executing, we may not be able to give every query execution its ideal number of servers. In this section we describe how to determine the ideal allocation of servers for a single query execution, and in Section 4 we examine how to balance the allocation across multiple executing queries. In this section, for clarity, we assume only a single partition is scanned on a server at a time, but our algorithm directly generalizes to support multiple (up to L_s) concurrent scans per server.

3.1 Overview

The goal of *adaptive server allocation* is to choose, for a given execution of a query, K_q: the number of servers at a time that should answer query q in parallel. Consider a query with a (possibly empty) predicate. Each server that is scanning a partition to answer part of the query will return data at a particular rate that depends both on the server hardware and on the selectivity of the predicate; a highly selective predicate will cause even a fast server to return data slowly. Similarly, the client can consume data at a given rate; this rate depends on the bandwidth capacity to the client as well as its own processing capacity and current load.

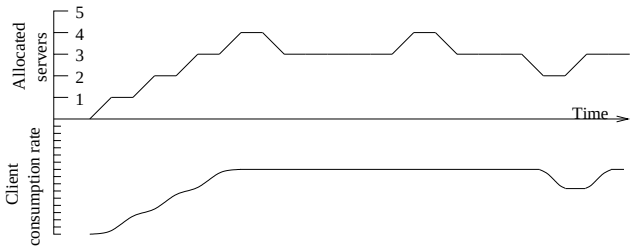

Figure 3: Idealized example of adaptive server allocation. The number of allocated servers increases until the client is saturated, and periodically increases or decreases thereafter to see if conditions have changed.

Figure 2 shows an example. Our goal is to assign enough servers to saturate the client's consumption capacity, but no more. Assigning more servers wastes resources without improving performance.

How do we choose the parallelism level K_q? Initially, our approach was to divide the client's maximum potential consumption rate by the average server sending rate. In practice, however, this approach did not work well. While we can measure the current client consumption rate, it is hard to measure the maximum rate at which the client *could* consume data. Because of burstiness caused by TCP windowing, packet queueing effects, and other factors, it is difficult to accurately measure how much throughput a client's network link can tolerate. Moreover, network capacity may not be the only bottleneck. A client machine may be overloaded processing business logic, rendering web pages or writing to its own disk. This load is highly variable, making it difficult to determine at a given point in time how much available capacity the client has. For these reasons, after repeated implementations we abandoned the rate-matching approach.

Instead, we adopted a simpler adaptive approach. We set K_q initially by increasing the allocation of servers until we observe that the client's consumption rate stops increasing. Of course, variations in load on the client, network and server, burstiness in the data distribution, and other factors mean that the ideal number of servers can change after we choose the initial K_q. Therefore, we must continually try to adapt K_q to current conditions. Periodically we flip a coin and decide to increase the number of servers allocated from K_q to $K_q + 1$. If increasing the number of servers results in an increase in the client consumption rate, we leave the allocation at $K_q + 1$ servers and perhaps allocate even more; otherwise we decrease it back to K_q. Similarly, we occasionally randomly decide to decrease the number of allocated servers to see if doing so negatively impacts the client's throughput. An idealized adaptation example is shown in Figure 3. In this way the system continually adjusts server allocation according to current system conditions.

Our adaptive approach is similar in spirit to the adaptive sizing of TCP windows for determining how much data can be in flight without causing network congestion. One difference in our method compared to the TCP algorithm is that TCP windows shrink by a multiplicative factor to quickly relieve congestion, while we only decrease K_q by one when deciding to revoke servers. In networks, congestion causes severe performance degradation since packets get dropped and must be retransmitted, making a sharp shrinkage in window sizes necessary despite performance impacts on the affected client. In our system, assigning too many servers slows performance for other queries, but the impact is not as severe, nor does it cause re-sending of data. However, revoking too many servers will cause a serious performance degradation for the client, and it will be hard to re-acquire those servers if there are other concurrent queries. Thus, we can afford to be less draconian when we have assigned too many servers. Note that the TCP approach is already used in our system for individual flows, since results are sent over TCP connections. ASA attempts to further improve the adaptive allocation of servers to queries, beyond the flow control and buffering benefits TCP provides.

Algorithm 1 Adaptive server allocation

Setting K_q initially:

1. Choose an initial number of servers $K_q(0)$
2. Assign $K_q(0)$ servers and begin scanning
3. Measure client consumption rate $C_q(0)$ for T_C seconds
4. Repeat
   (a) Set $K_q(i+1) = K_q(i) + 1$
   (b) Assign another server and begin scanning
   (c) Measure client consumption rate $C_q(i+1)$ for T_C seconds

   While $C_q(i+1) > C_q(i)$

Adapting K_q:

5. Every T_{flip} seconds, flip a coin:
   (a) With probability $P_\uparrow$, set $K_q(i+1) = K_q(i) + 1$
      i. Allocate another server
      ii. Measure client consumption rate $C_q(i+1)$ for T_C seconds
      iii. If $C_q(i+1) > C_q(i)$, set $K_q(i+2) = K_q(i+1) + 1$ and return to step 5.a.i to allocate another server; otherwise, set $K_q(i+2) = K_q(i+1) - 1$ (revoke a server)
   (b) With probability $P_\downarrow$, set $K_q(i+1) = K_q(i) - 1$
      i. Revoke a server
      ii. Measure client consumption rate $C_q(i+1)$ for T_C seconds
      iii. If $C_q(i+1) \geq C_q(i)$, set $K_q(i+2) = K_q(i+1) - 1$ and return to step 5.b.i to revoke another server; otherwise set $K_q(i+2) = K_q(i+1) + 1$ (give back a server)

3.2 Allocation algorithm

More formally, our system allocates servers to a particular query execution as shown in Algorithm 1. In this algorithm, T_C is the interval over which we measure the client consumption rate. T_C should be long enough to allow the client to begin noticing the effects of the new allocation and to smooth out noise, but not so long as to hinder adaptivity. The interval between coin flips, T_{flip}, and the probabilities of re-allocation, $P_\uparrow$ and $P_\downarrow$, ($P_\uparrow + P_\downarrow \leq 1$) determine how aggressively we try to re-adapt after reaching client saturation. In practice, T_C and T_{flip} set to one second, $K_q(0)$ set to one, and $P_\uparrow = P_\downarrow = 1/3$ work well.

In our implementation when we revoke a server (steps 5.a.iii and 5.b.i), we do not immediately interrupt a partition scan on a server. This simplifies our implementation. Instead, we wait for a partition scan to complete; but when

we are done with a server we do not assign another server to the query. This effectively revokes a server. Additionally, we wait to measure the new client consumption rate in step 5.a.ii until after the newly allocated server from 5.a.i actually begins scanning; similarly, after revoking a server in 5.b.i, we wait until a server finishes and is effectively revoked before measuring the new client consumption rate in step 5.b.ii.

4. SERVER SCHEDULING

While adaptive server allocation caps the number of storage servers concurrently granted to each query, the scheduler coordinates which servers are given to a query. When a server becomes free, the scheduler grants that server to a query that wants it, ensuring that no server s has more than L_s concurrent scan requests. When server demand is high, and it is impossible for each query to get its K_q servers, the scheduler additionally must prioritize queries, giving more servers to some queries and fewer to others.

In this section we first describe our scheduling mechanism. We then show that a greedy scheduler will bound the *makespan* (time to complete all queries). Given this foundation, we describe how our *Priority Biased Round Robin* (PBRR) metric lets us prioritize queries based on different, pluggable criteria, and we provide a greedy heuristic for optimizing this metric.

4.1 Scheduler Component

The scheduler runs alongside the scan engine and maintains state on the set of current queries $\mathbf{Q}$ and storage servers $\mathbf{S}$. When a query is submitted, its scan engine process initially contacts the scheduler with a list of requested servers and partitions. All further communication is through two message types. The scheduler sends a *grant* message to a scan process to notify it to scan a server on behalf of its query. A scan process sends a *relinquish* message to the scheduler when it has finished with a server. Although our scheduler can deal with multiple replicas of data partitions (choosing which replica to use based on concurrent load) for clarity in the discussion below we assume that there is only a single copy of each data partition.

4.2 Greedy Scheduling Algorithms

Makespan measures the time to complete an entire query workload. We evaluate a scheduling policy by the upper bound on its makespan, or *maximum makespan,* as a multiple of the optimal achievable makespan for the workload. A policy with a bounded worst-case makespan ensures that we complete the workload in a reasonable amount of time and thus do not starve any queries. Of course, in a real deployment new queries will constantly enter the system and the workload will never complete. However, a makespan-minimizing scheduler will ensure whatever queries are in the system at a given point in time are making good progress.

Greedy algorithms are an effective approach for minimizing makespan for many scheduling problems [19], and are typically efficient to execute online. For these reasons, we implemented our scheduler using a greedy scheduling policy. Whenever a server becomes free, our scheduler grants it to some query that (a) requests it and (b) is below its K_q. If multiple queries fulfill these criteria, the scheduler greedily chooses the "highest priority" query. The definition of "highest priority" determines our scheduling policy. For example, if priority is equal to query start time, the scheduler is FIFO; if priority is equal to our PBRR metric (defined in Section 4.3), the result is our priority-based scheduler.

We now present our system model and prove that any greedy algorithm bounds maximum makespan.

4.2.1 Model

Let $\mathbf{Q}$ be a set of queries and $\mathbf{S}$ be the set of storage servers. A query $q \in \mathbf{Q}$ consists of a set of unit-size partition scan *jobs* $j \in q$, each of which should be executed on a designated server $s_j \in \mathbf{S}$. Jobs may be executed out of order, i.e., there are no precedence constraints (though in future work we may add ordering constraints). Each query q has an associated parallelism level K_q, the maximum number of jobs from that query that may be executed concurrently on the servers in S. K_q is determined using adaptive server allocation (Section 3.2). For modeling purposes, we assume K_q remains constant through the execution of the query. Each server s has an associated number L_s that denotes the number of jobs that can be concurrently executed on s.

Let $q_j \in \mathbf{Q}$ denote the query containing job j, and j_s denote the set of jobs s must execute, i.e.,
$j_s = \left\{ j \in \bigcup_{q \in \mathbf{Q}} q : s_j = s \right\}$.

4.2.2 Greedy Maximum Makespan

Let OPT denote the minimum makespan for a problem instance. Define $\hat{q} = \max_{q \in \mathbf{Q}} \left\lceil \frac{|q|}{K_q} \right\rceil$ as the maximal time to execute any one query at full parallelism. Define $\hat{\jmath} = \max_{s \in \mathbf{S}} \left\lceil \frac{|j_s|}{L_s} \right\rceil$ as the maximal time for any one server to complete all of its jobs at full concurrency. We first prove a tight bound on OPT.

THEOREM 1. $\text{OPT} = \max\{\hat{q}, \hat{\jmath}\}$.

PROOF. We first show that $\text{OPT} \geq \max\{\hat{q}, \hat{\jmath}\}$. It suffices to separately prove $\text{OPT} \geq \hat{q}$ and $\text{OPT} \geq \hat{\jmath}$. To prove $\text{OPT} \geq \hat{q}$, consider a query q that has a full parallel completion time of $\hat{q}$. At any given time slot, there can be at most K_q servers executing jobs from q. Thus to complete q, at least $\lceil |q|/K_q \rceil = \hat{q}$ time slots are required in a schedule.

To prove $\text{OPT} \geq \hat{\jmath}$, consider a server s with $\hat{\jmath}$ jobs to be run on it. Since s can execute at most L_s jobs at a time, it will not complete the workload until after at least $\lceil |j_s|/L_s \rceil = \hat{\jmath}$ time slots.

We now prove that $\text{OPT} \leq \max\{\hat{q}, \hat{\jmath}\}$. Consider the bipartite undirected multigraph (without self-loops) $G = (\mathbf{Q}, \mathbf{S}, \mathbf{E})$, where $\mathbf{E} \subseteq \mathbf{Q} \times \mathbf{S}$ contains edges of the form (q, s) if and only if $s_j = s$ for some $j \in q$. Note that if multiple $j \in q$ satisfy $s_j = s$, there will be multiple (q, s) edges in $\mathbf{E}$. Define $f : \mathbf{Q} \cup \mathbf{S} \to \mathbb{N}$ such that $f(q) = K_q$ for $q \in \mathbf{Q}$ and $f(s) = L_s$ for $s \in \mathbf{S}$. An *f-edge coloring* assigns an integer (color) to each edge in $\mathbf{E}$ such that no more than $f(v)$ edges incident with $v \in \mathbf{Q} \cup \mathbf{S}$ have the same color. Let $d(v)$ for $v \in \mathbf{Q} \cup \mathbf{S}$ denote the *degree* of v. By Theorem 5 in Zhou et al. [25], an f-edge coloring for G can be found that uses at most

$$\begin{aligned} \max_{v \in \mathbf{Q} \cup \mathbf{S}} \left\lceil \frac{d(v)}{f(v)} \right\rceil &= \max \left\{ \max_{q \in \mathbf{Q}} \left\lceil \frac{d(q)}{K_q} \right\rceil, \max_{s \in \mathbf{S}} \left\lceil \frac{d(s)}{L_s} \right\rceil \right\} \\ &= \max \{\hat{q}, \hat{\jmath}\} \text{ colors.} \end{aligned}$$

Each edge in G corresponds to an assignment of job to a server, so edges with the same color in the f-edge coloring can be scheduled simultaneously. By the construction

of f, no more than K_q jobs from the same query q will be scheduled concurrently, nor will any server s run more than L_s jobs in parallel. Combining both proven inequalities, the makespan of this schedule is thus equal to the total number of colors, $\max\{\hat{q}, \hat{j}\}$, which proves the theorem. □

We next prove an upper bound on the maximum makespan of greedy algorithms in our model.

THEOREM 2. *The makespan of any possible greedy algorithm is no more than twice the makespan of* OPT.

PROOF. Let server s have the highest makespan r among all servers in the schedule produced by a greedy algorithm, such that $r > \hat{j}$. We will show that $r \leq 2\max\{\hat{q}, \hat{j}\}$. Let ℓ be a job scheduled on s at the final time r, and assume ℓ belongs to query q. Recall that j_s is the set of jobs that must be executed on s. There are at least $r - \lceil |j_s|/L_s \rceil$ *gaps*, or times during which s did not fill all L_s slots with jobs. The only reason for ℓ not to have been scheduled in one of these gaps before r is that for each gap, there must have been K_q other jobs from q simultaneously scheduled in the system. Otherwise, any greedy algorithm would have scheduled ℓ in one of these gaps.

Because during each such gap there must be K_q jobs being executed for q, the number of gaps can be at most $|q|/K_q \leq \hat{q}$. Thus, $r - \lceil |j_s|/L_s \rceil \leq \hat{q}$ so $r \leq \hat{q} + \hat{j}$. By Theorem 1, $\text{OPT} = \max\{\hat{q}, \hat{j}\}$. We conclude that

$$\text{GREEDY} = r \leq \hat{q} + \hat{j} \leq 2\max\{\hat{q}, \hat{j}\} = 2 \cdot \text{OPT}. \quad \square$$

4.3 Priority-Biased Round Robin

Theorem 2 ensures that any greedy algorithm makes good progress on all queries in a workload (e.g., maximum makespan is bounded.) We now present a greedy scheduling algorithm that additionally allows us to prioritize certain classes of queries. Two of the many possible criteria are:

- **Gold vs. best-effort clients**: Some clients are considered more important than others. Gold clients, for example, have paid extra to guarantee fast response time, even when the system is loaded.
- **Short/fast vs. long/slow queries**: Short queries scanning one or a few partitions from a fast client typically are a customer-facing request, where a fast response is important. Long queries scanning many or all partitions from a slow client may be a batch process, where response time is less urgent (but still important).

The scheduler will prioritize queries based on our chosen criteria. At the same time, it will ensure that all queries make progress, and we prefer steady progress to bursty progress with intermittent gaps of no results. The reason is that the client must process the query results in some way, either displaying them to an end user, or doing further computation. Bursty results make processing or presenting results uneven and wastes client resources.

Our *Priority Biased Round Robin* (PBRR) metric, which the scheduler minimizes, trades off between the goals of prioritization and minimizing idle times between results. For the purposes of the metric, each query q has a priority value, $P(q) > 0$, with a lower value indicating higher priority. The priority term encapsulates our scheduling preference (e.g., favoring short versus long queries or gold versus best-effort clients) and we discuss setting this shortly. Each q also has a set of *idle times*, $I(q)$, which is the set of durations over which q was not scheduled on any server. Finally, we have a scheduler-wide parameter α that controls the relative importance of priority versus idle periods when choosing between queries. To compute PBRR for a set of queries $\mathbf{Q}$, for each q we sum the squares of the idle times and divide by priority raised to the α power. Then, we sum over all q's:

$$\text{PBRR} = \sum_{q \in \mathbf{Q}} \frac{C + \sum_{i \in I(q)} \text{length}(i)^2}{P(q)^\alpha} \tag{1}$$

We square the idle time term length(i) to favor many short idle periods over a few long ones. C is a small constant (e.g., $C = 1$) that ensures the numerator is non-zero when there are no queries with idle times; in that scenario, the priorities in the denominator determine which queries get scheduled.

The parameter α is key to tuning the behavior of our scheduler. It is the knob that controls the weight given to $P(q)$ relative to the idle times. An α setting greater than 1 magnifies $P(q)$'s impact and diminishes the impact of idle times. $\alpha = 0$ essentially sets all priorities to 1. A negative α actually reverses the priorities.

Minimizing this metric captures our scheduling preferences. The higher priority a query has, the more penalty we pay when it is idle; since additions to the numerator are magnified by the relatively smaller denominator of high priority queries. At the same time, no query will starve: as an idle duration for any query grows linearly, the penalty toward the metric grows quadratically, rapidly increasing the likelihood that the query is scheduled.

4.3.1 Setting Priority

We now discuss how to set the priority $P(q)$ variable in Equation 1. Consider the scenario where we want to support gold and best-effort priority levels, where queries come tagged with some externally bestowed priority value. We simply set $P(q)$ to this value. Then, the relative priorities of other queries and α impact the level of preferential treatment a particular priority setting gives.

Next consider a scenario where we want to favor short or long queries. We set $P(q)$ to the query's *minimum completion time*, $MC(q)$. This value denotes the time in which q can complete, if given top priority for all servers. The following lemma tells us how to compute this time.

LEMMA 1. *The minimum completion time $MC(q)$ for query q equals* $\max\{\hat{q}, \hat{j}\}$.

PROOF. We observe that $MC(q)$ is a special case of finding OPT where q is the only query to be scheduled. Theorem 1 proves the tight bound on OPT, and therefore proves a tight bound on $MC(q)$. □

4.3.2 Implementing the PBRR Scheduler

Our priority-based scheduler, which schedules queries to minimize the PBRR metric, periodically runs Algorithm 2. The scheduler stores priority function P and tracks the current idle duration for each query. Assume query q has been idle for i_q time units. We utilize a priority queue, Π, and insert a query into it with value $\frac{C+i_q^2}{P(q)^\alpha}$, the penalty to PBRR if the query is not scheduled.

We call attention to two special cases. Consider the case where all queries are non-idle. When we schedule a query q, $P(q)$ does not change. Therefore, the highest priority query

Algorithm 2 PBRR scheduling

1. For each q, insert q into Π with value $\frac{C+i_q^2}{P(q)^\alpha}$.
2. While servers are idle or Π non-empty
   (a) Pop highest priority query q^* from Π.
   (b) If q^* has K_{q^*} servers, set q^* aside.
   (c) If q^* cannot use any idle servers, set q^* aside. Else
      i. Grant q^* one idle server.
      ii. Set $i_{q^*} = 0$.
      iii. Insert q^* into Π with value $\frac{C+i_{q^*}^2}{P(q^*)^\alpha}$.

at the start of the scheduling instance remains highest priority throughout; a lower priority query can only be scheduled once all queries above it reach K_q or do not need any of the idle servers. This is, in fact, the behavior of a priority scheduler. The second case is when $\alpha = 0$; then $P(q)^\alpha = 1$ and the highest priority query is that with maximum current idle time. The scheduler reduces to a round-robin scheduler.

5. EVALUATION

We now examine the impact of parallelism in a distributed data store, and evaluate our techniques for adaptive server allocation and scheduling. We have run experiments using both synthetic data and a data set from Flickr's database of photo metadata. At a high level:

- Our results verify the intuition that a query benefits more from parallelism when it is longer, more selective, or coming from a client with faster consumption rate (server speed is also a factor but tends not to vary from query to query). For example, two of our storage servers provided enough parallelism to saturate a slow (500 KB/second) client, while eight servers were needed to saturate a faster (2,000 KB/second) client.
- Adaptive server allocation (ASA) tunes a query's K_q value to get good throughput, and outperforms policies that assign all queries a fixed K_q. For example, in a multi-query setting with slow clients, ASA beats allowing all clients to request all servers by a factor of 2 to 3.
- Our PBRR scheduler gives us the ability to prioritize queries. PBRR does much better than FIFO in terms of time until each query receives its first results, as well as delivering steady results. By tuning α we are able to clearly prefer short queries over long and gold queries over best-effort.

We also examine the effect of range queries on concurrent point lookups.

5.1 Experimental Setup

We augmented the production PNUTS code by implementing our range processing components (the scan engine and scheduler). We set up a cluster with 10 storage servers, four client machines, a router and a partition controller. Our load generation application used the PNUTS client library, which we augmented with instrumentation to measure its consumption rate (KB/second) and periodically report this value to the scan engine to support ASA. All machines run Red Hat Enterprise Linux 4.3 and have a single disk. We use the Trickle utility [16] to precisely control clients' consumption speeds for our experiments.

The experiments reported in this paper used synthetic data so that we could have precise control over key distribution and query selectivity. We created a table of 1024 partitions, and loaded 20,000,000 5KB records distributed evenly among the partitions. This places roughly 100 100MB partitions on each server, for roughly 10 GB total on each server. Each record also has an integer *pred* field whose value is chosen randomly and uniformly from the range 1-1,000. Our range queries have an inequality selection predicate on *pred* that we use to control query selectivity. The storage servers filter out records not passing the predicate. In the rest of this section we refer to selectivity by the percentage of records passing the predicate.

To validate our results, we also ran experiments using a data set from Flickr representing metadata for 10 million Flickr photos, ordered by date. The average record size was 214 bytes. Our range queries represented a workload of finding photos from specified time intervals which matched a predicate over the photo's ranking. This workload is similar to a common user activity of trying to find highly ranked photos. Time intervals in our queries were randomly distributed across the range of time represented by the records. The rankings displayed no apparent correlation to date. Since we did not have a trace of prioritized Flickr queries (Flickr does not have queries with varying priorities) we did not run scheduling experiments using this data. Results from these experiments are not reported here, but were consistent with the reported results. See [2] for full results.

5.1.1 *Setting the per-server concurrency limit (L_s)*

We ran experiments to determine how many concurrent range scans we could assign to each storage server. Our results are as expected: limiting contention improves scan time, because sequential I/O can be used. On a server with one disk, limiting scans to one at a time allowed all scans to complete 14 percent faster than allowing two at a time, and 50 percent faster than allowing 32 at a time. Similarly, with two disks, two concurrent scans allowed all queries to complete 50 percent faster than only scan at a time, and 23 percent faster than allowing three scans. For the rest of our experiments, we use single-disk servers and $L_s = 1$. Note that L_s represents a tradeoff between throughput and latency: higher values can reduce latency to returning the first result by allowing more outstanding concurrent requests, at the cost for poorer overall throughput due to head contention.

5.2 Parallelism Impact

We now examine the variables we have identified as key to determining the best parallelism: query length, query selectivity, and client speed. In these experiments, we run a single query at a time, vary the parallelism level, and measure the time to complete the query. With $L_s = 1$ and ten servers, maximum parallelism is ten.

Varying query length - Figure 4 shows the effects on query completion time of varying query length from 0.5% to 4% of the table, with selectivity fixed at returning 10% of tuples scanned, and client speed fixed at 10,000 KB/sec (making queries server-bound). The results show that as query length grows, high parallelism is increasingly important for achieving a fast completion time. In addition, while increasing parallelism causes a drop in response times, the benefits diminish as K_q continues to grow. Diminishing returns are

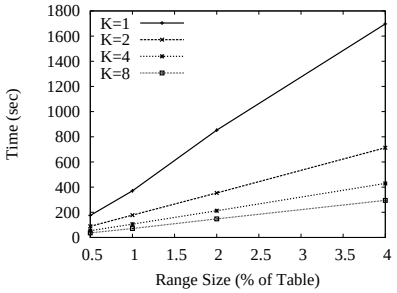

Figure 4: Range size vs. completion time.

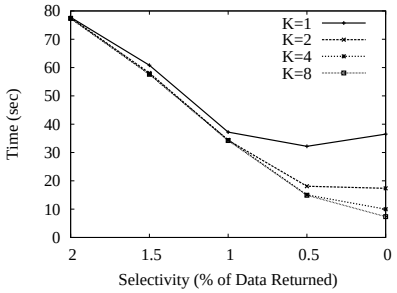

Figure 5: Query selectivity vs. completion time.

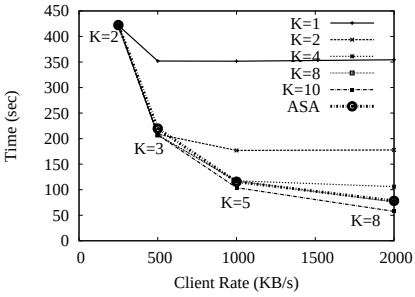

Figure 6: Client speed vs. completion time.

common in parallel systems; in our case they are caused by varying performance among the servers, and the query not being completely evenly distributed across the servers. Our key observation is that as query length increases, the greater parallelism's impact is on overall running time.

Varying query selectivity - Figure 5 shows the effect of varying query selectivity from returning 2% to 0% of tuples scanned. Length is fixed at 1% of the table and speed fixed at 250 KB/sec, so the experiment is client-bound. At lower selectivities, we see no distinction between different parallel levels. $K_q = 1$ returns tuples quickly enough to saturate the client, and higher parallelism results in no gain. At 1% we see $K_q = 1$ peel away from the other plots; this is the point where it takes 2 servers to saturate the client. We see a similar effect at 0.5%, where it takes 4 servers. In general, the fewer results a server generates, the more a client benefits from parallelism, even if the client is slow.

Varying client speed - Figure 6 varies client speed from 250 to 2,000 KB/sec. Length is fixed at 1% of the table and selectivity is fixed at returning 10% of tuples scanned. We again plot different parallel levels. For now, ignore the ASA adaptive server allocation plot and its associated K_q labels; we will return to this shortly. We again see a peeling away pattern. At 250 KB/sec, one server is sufficient to saturate the client; at higher parallel levels, results are buffered between the range processor and client. At 500 KB/sec, $K_q = 1$ peels away and 2 servers are required for saturation. Likewise, 4 servers are needed at 1,000 KB/sec, and 8 are required at 2,000 KB/sec. We conclude that (a) fast clients benefit more from parallelism than slow clients, and (b) there is no benefit to granting a client servers beyond the amount necessary to saturate it.

5.3 Adaptive Server Allocation

We now examine the effectiveness of adaptive server allocation (ASA). Recall from Section 3.2 that our adaptive algorithm works by increasing K_q until the client stops increasing its consumption. Then, the algorithm periodically adjusts K_q up or down, based on probabilities $P_\uparrow$ and $P_\downarrow$. From experimentation, we found that the probability values did not have a major influence on the algorithm's ability to find a good K_q; thus, we will use $P_\uparrow = P_\downarrow = 1/3$.

5.3.1 ASA – Single Query

We return to Figure 6 and consider the ASA plot. ASA achieves roughly the same performance as $K_q = 10$, without allocating all ten servers to the query. The K_q annotations on the graph show the actual number of servers allocated to the query. Although ASA performs well, it does not quite achieve the optimal result in two ways. First, the performance is slightly slower (up to 31% in some cases) than $K_q = 10$. Second, in some cases one less server could have been assigned without losing performance; e.g., at 1,000 KB/sec, $K_q = 4$ would have worked and ASA chooses $K_q = 5$. Both effects result from adaptively allocating or revoking servers, and the non-zero time necessary to measure the effects of a new allocation or wait for a server to finish scanning so it can be revoked. We argue that some sub-optimality as the result of using an adaptive heuristic is an acceptable price for not having to hand-tune the system.

We can see the behavior of ASA in Figure 7 which plots, for the case of client rate of 1,000 KB/sec, a snapshot of experiment time vs. requested and actual K_q taken from the first 60 seconds of the experiment. Early on, ASA requests $K_q = 5$ and the scheduler quickly grants the request. The client consumption rate does not increase from having the fifth server, however, and ASA lowers its request to $K_q = 4$. The fifth server continues scanning, and approximately 45 seconds into the query, actual K_q finally drops to 4. Shortly after, we see another test by ASA of $K_q = 5$. In this case, thanks to another of the original 5 granted servers finishing, actual K_q quickly drops back to 4. This graph illustrates the effect discussed above; although we may decide to change server allocation, the allocation itself may not immediately change. We could interrupt server scans to deal with this issue, but we must examine in ongoing work whether the performance benefit is worth the added complexity.

Overall, we conclude ASA is effective for automatically finding an appropriate K_q setting for different client speeds.

5.3.2 ASA – Multi Query

When there are multiple queries, ASA is key to ensure we do not allocate too few or too many servers to one query. The scheduler also becomes important to mediate between queries. In this section, we focus on ASA; we use a FIFO scheduler and return to the scheduler itself in Section 5.4.

Multiple queries, same client speeds - First, we examine whether a fixed policy, or our adaptive ASA, is the best way to set K_q when all clients have the same speed. Figure 8 depicts an experiment identical to our previous one (query scans 1% of table range, returning 10% of scanned tuples), but now with 4 such queries each requesting a different random range. Each query is issued by a different client machine. Clients share the same consumption rate.

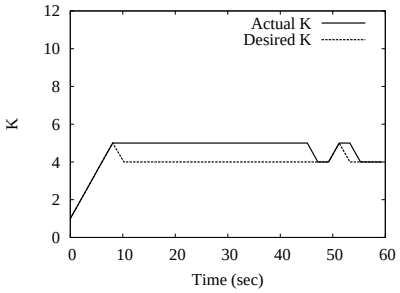

Figure 7: Time vs. K_q, client speed 1,000 KB/sec.

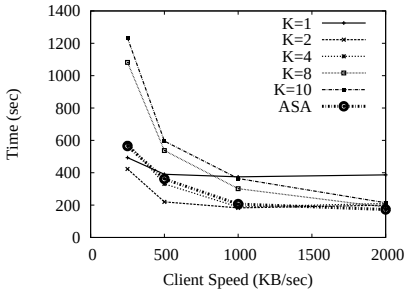

Figure 8: Client speed vs. workload completion time, 4 query workload.

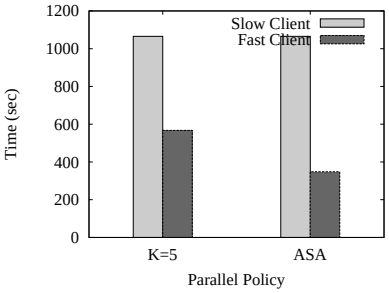

Figure 9: Query completion times for $K_q = 5$ and ASA.

As the figure shows, allocating many or all of the servers to each query (e.g., $K_q = 10$) hurts performance when clients are slow, since one query grabs lots of servers, starving the other queries. In contrast, allocating just one server ($K_q = 1$) causes poor performance when clients are fast, since each client could benefit from more parallelism. The best policies across all client ranges are ASA and $K_q = 2$. It makes sense that $K_q = 2$ performs well; if we divide the 10 servers roughly equally among all queries, each would receive 2 servers. Thus we might ask whether a static policy that sets $K_q = \frac{(num\ servers)}{(num\ clients)}$ is sufficient. There are two problems with this approach. First, the number of clients changes as queries are submitted or completed, making it difficult to choose the right K_q. Second, even if the number of clients remained fixed, when clients have different speeds, evenly dividing the servers results in poor performance. We examine this effect in the next section. Thus, we conclude that ASA is more effective compared to fixed K_q policies.

Multiple queries, different speeds - We examine performance when clients have different speeds. We ran an experiment with two clients, one which could consume 100 KB/sec (the *slow* client) and another which could consume 2,000 KB/sec (the *fast* client). Each client submitted one range query; both queries scanned a different 1% of the table with a predicate that returned 10% of scanned tuples. Figure 9 shows a static policy of dividing the 10 servers equally between the two clients ($K_q = 5$) and our ASA policy. As the figure shows, the slow client is client-bound, and completes in the same time in both cases. In contrast, for the fast client, ASA is faster than $K_q = 5$. Under the $K_q = 5$ policy, we grant 5 servers to a slow client that cannot make use of more than 1 or 2. ASA grants only 1 or 2 servers to the slow client, and therefore has more servers available to grant to the fast client. In theory we could have a fixed policy that manually set a query's K_q based on measured client speed; this is in fact what ASA does automatically.

The fast client does not perform as well as it could if it were the only query executing, since it occasionally must wait for a server currently allocated to the slow client. Again, interrupting server scans may be useful to further improve performance, and this is a topic of future work. Overall, ASA works better than dividing servers evenly among queries when queries have different consumption rates.

5.4 Scheduling

We now focus on scheduling in the multi-query environment and specifically consider cases where it is impossible to grant all queries their ASA-requested K_q. The problem is to prioritize which queries are scheduled. As discussed in Section 4 our scheduler lets us prioritize on various criteria, while also attempting to provide a steady stream of results to all clients. We compare the following schedulers:

- FIFO - earliest query gets top priority.
- Random - free server is assigned to a random query.
- Round Robin (RR) - longest idle query gets top priority.
- PBRR - our scheduler described in Section 4.3.2.

ASA is used in all experiments in this section.

Priority based on length - We first base priority on query length. We construct queries that are random ranges with lengths 1,2,4 and 8% of the table, all returning 10% of tuples scanned. There are 4 clients and each generates one query of each length, for a total workload of 16 queries. All queries are issued immediately. To avoid biasing FIFO, each client issues its queries in different orders.

Figure 10 plots results for different scheduler settings. For each setting, we show a cluster of four bars, where each bar shows aggregate results for a particular length query; per cluster, the bars are ordered from smallest to longest length. For example, the first bar of the first cluster depicts FIFO scheduler with length 1% queries. Each bar shows average time until first results are returned, average completion time, and maximum completion time.

The major weaknesses of FIFO are clear. Most queries wait a long time before getting any results. Prioritization is completely at the mercy of when the queries hit the scheduler; time to first result is independent of query length. Average completion time trends with query length, but the differences are small. Finally, maximum completion time is independent of query length.

Contrast this with the other schedulers, which all look similar to round-robin. Every query gets some results relatively quickly, and shorter queries tend to finish faster. With our PBRR scheduler, we bias these trends. With positive α values, we favor shorter queries, and they get their first results earlier. With negative α values, we do the opposite. Round-robin, which is in fact $\alpha = 0$, is oblivious to length.

To visualize the pace at which different length queries get results, we plot in Figure 11, for the $\alpha = 2$ case, *time versus KB received* for one length 1% query and one length 8% query. This plot covers the beginning portion of the experiment. We see that length 1% gets results more quickly and with shorter periods of no results. Likely the shorter query is client-facing, while the longer is a batch process, and this

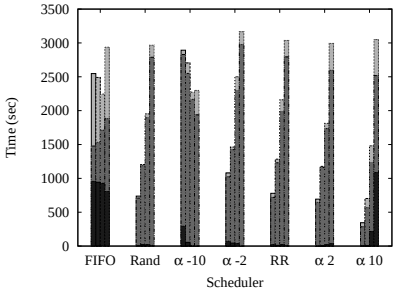

Figure 10: Workload performance by scheduler, length-based priority. A cluster shows aggregate results for length 1,2,4,8% queries. A bar shows first result, average completion, max completion times.

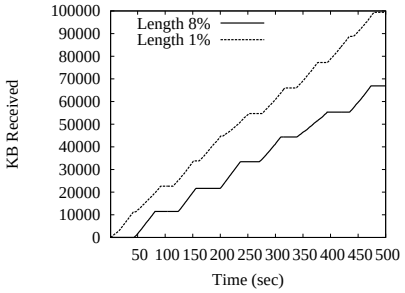

Figure 11: Time vs. KBs for 2 queries of lengths 1% and 8% from Figure 10, $\alpha = 2$ setting.

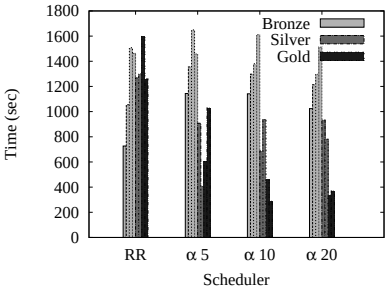

Figure 12: Workload performance for different schedulers, priority defined externally. Each bar cluster shows completion time for each query.

is an appropriate scheduler setting. We see that using the PBRR scheduler, we can effectively favor short queries over long, but deliver all clients a steady stream of results.

Externally bestowed client priorities - We ran a similar experiment with priority based on client service level. In this setting, there are 8 queries and three priority levels. Two queries have a "gold" level of priority 1, two have "silver" priority 2, and four have "bronze" priority 4. All queries are length 4% and return 10% of tuples scanned. The queries are issued slightly staggered such that they arrive at the scheduler in inverse-priority order. This allows us to see if we can tune our scheduler such that a high priority query performs well despite an earlier arriving low priority query. Figure 12 shows the completion time for each of the 8 queries for different scheduler settings. Each cluster of bars is ordered by query arrival time at the scheduler. With $\alpha \geq 10$ the gold queries clearly get top priority. We note two interesting features. First, the very first query gets a significant advantage from its head start on acquiring servers; even at $\alpha = 20$, its completion time is only slightly higher than the slowest silver query. Second, the workload completion times in this experiment and in Figure 10 are both unaffected by α setting, as we would expect from Theorem 2.

5.4.1 Scheduler at Scale

In order to test our scheduler at larger scales, we built a stand-alone discrete time-based simulator. It generates synthetic workloads, maps queries to servers by using one of the scheduling heuristics, simulates network and disk scan delays when retrieving results, and measures the makespan for the given schedule. For simplicity, we assume that the level of parallelism of a query is fully known at the time it is dispatched, and that the results are uniformly distributed over the partitions requested. We assume that each server can execute at most one query at a time.

We ran a simulation to repeat the experiment from Figure 12, $\alpha = 10$ case and proportionally scale up the number of servers and queries from 10 servers and 8 queries (as in the actual experiment) up to 200 servers and 160 queries (ratio of gold/silver/bronze queries remains constant). The result is straightforward, and we omit the plot. At 10 servers and 8 queries, the bronze queries finish in average of 1,200 seconds, the silver queries in 700 seconds, and the gold queries in 400 seconds. As we increase the number of server and queries, these average completion times stay constant, verifying that our scheduler results hold at larger scales.

5.5 Impact on Normal Traffic

In practice range queries are not the only type of queries in the system. In fact, the majority of queries are likely point queries, each requesting a single record. We investigated the impact on point query performance by running a point query workload by itself, and then by running the same workload concurrent with range queries. In the latter case, we ran four range queries, each at $K_q = 10$, and made sure that every server was processing range requests throughout the point query workload. Any slowdown caused by the range queries affected all point queries equally. In the absence of range queries, median point query latency was 5 ms, with 22% of queries taking longer than 10 ms. With range queries, median latency held at 5 ms, but 28% of queries took longer than 10 ms. We see then that range queries have little adverse effect on most point traffic, but do increase the number of queries that experience longer latency.

6. RELATED WORK

Shared nothing databases - Shared nothing databases that can scale out by adding hardware have existed for decades [15]. Our system builds on many of the concepts presented in this earlier work, including range partitioning and intra-operator parallelism. Our approach of limiting the number of servers for a query is complementary to the per-operator flow control of [15].

Previous work assumed maximum parallelism would be given to queries, and focused on maximizing throughput by placing [7, 17], moving [18] or copying [21] partitions. Our approach of determining how many partitions to scan in parallel is complementary to partition placement techniques.

Massive scale systems - Several massive-scale database systems have been built recently [1, 4, 10, 13, 20]. While some of these support executing range queries in parallel [10, 20], their published works do not describe mechanisms to optimize parallelism level, and they could possibly benefit from our techniques. Other systems are hash tables and do not support range queries (such as Dynamo [13]).

Column-stores are optimized specifically for scan tasks, es-

pecially over read-mostly data [23]. Our adaptive parallelism techniques could be adapted to work on a parallel column store. Moreover, our techniques work well even for frequently updated data. MapReduce [12] also focuses on scanning for analysis queries, and carefully exploits parallelism. However, adaptivity is not typically needed because the number of concurrent scans for a MapReduce job is statically set to the number of map tasks that have been spawned.

An earlier approach to building scale-out systems was to build networks of workstations [6]. Work on such systems focused on how to achieve maximum parallelism, rather than throttling or scheduling parallelism to satisfy multiple clients and their constraints. The Condor scheduling system [24] allocated servers to parallel jobs to minimize CPU contention. Our problem is more constrained, since data is harder to move than computation.

Other work in parallel databases - Earlier work in parallel databases [14, 9] describes maximizing the amount of parallelism for queries, and does not focus on isolating the performance of concurrent queries or limiting the amount of parallelism. Commercial systems provide parallel scanning as well as mechanisms for tuning [3, 5, 8]. For example Oracle provides parameters to manually limit the amount of parallelism, and implements a fixed allocation policy ($K = \frac{(num\ servers)}{(num\ clients)}$). Our evaluation results show the benefit of more adaptive techniques.

Scheduling - The scheduling problem in Section 4 to minimize makespan is similar to the *open shop environment* with identical processing lengths, particularly $O|p_{ij} = 1|C_{\max}$ (in scheduling notation) [19]. Determining the minimum makespan in that setting can be done in almost linear time in the number of jobs using a bipartite edge-coloring algorithm [22], where jobs with the same color can be scheduled concurrently. There is a key difference, however, between our problem and the standard open shop environment. In ours, each job (query) i can have up to K_i concurrent operations, and each processor (storage server) j can execute up to L_j parallel operations. In this more general case, as described in Theorem 1, a schedule with minimal makespan can be found in polynomial time using [25]. It is unclear how this algorithm works on weighted graphs, where edge weights correspond to PBRR cost.

7. CONCLUSIONS

Web applications need database systems to provide high performance range scans for a variety of tasks. Parallel execution of range queries is key to achieving this high performance. However, if we assign too many servers to a query, we will waste resources and reduce the performance of other queries. We have described techniques for determining how many, and which, servers should be used in parallel to execute range queries in a massive scale shared nothing database. Our experimental results show that our adaptive server allocation algorithm can effectively determine how many servers are needed to satisfy a given client, even without prior knowledge of the data distribution or server and client characteristics. We have also shown that our adaptive scheduler can effectively assign servers to query executions to ensure good performance for all queries. Furthermore, our scheduler allows us to prioritize certain queries over others, based on our applications and policies. These techniques are key to making the best use of a large number of parallel resources when executing range queries.

8. REFERENCES

[1] Amazon SimpleDB. aws.amazon.com/simpledb/.

[2] Extended version. Yahoo! Labs Technical Report YL-2009-003, research.yahoo.com/Technical_Reports.

[3] Oracle Database Data Warehousing Guide, 10g release 2. download-west.oracle.com/docs/cd/B19306_01/-server.102/b14223/usingpe.htm.

[4] SQL Data Services/Azure Services Platform. www.microsoft.com/azure/data.mspx.

[5] Teradata. www.teradata.com.

[6] A. C. Arpaci-Dusseau et al. High-performance sorting on networks of workstations. In *SIGMOD*, 1997.

[7] M. J. Atallah and S. Prabhakar. (Almost) optimal parallel block access for range queries. In *PODS*, 2000.

[8] C. Baru et al. An overview of DB2 Parallel Edition. In *SIGMOD*, 1995.

[9] H. Boral et al. Prototyping Bubba, a highly parallel database system. *IEEE TKDE*, 2(1), March 1990.

[10] F. Chang et al. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[11] B. F. Cooper et al. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB*, 2008.

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

[13] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[14] D. J. DeWitt et al. The Gamma database machine project. *IEEE TKDE*, 2(1):44–63, 1990.

[15] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database processing. *CACM*, 36(6), June 1992.

[16] M. A. Eriksen. Trickle: A userland bandwidth shaper for Unix-like systems. In *Proc. USENIX Annual Technical Conference*, 2005.

[17] H. Ferhatosmanoglu, D. Agrawal, and A. E. Abbadi. Concentric hyperspaces and disk allocation for fast parallel range searching. In *ICDE*, 1999.

[18] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *VLDB*, 2004.

[19] D. Karger, C. Stein, and J. Wein. *Handbook of Algorithms and Theory of Computation*, chapter Scheduling Algorithms. CRC Press, 1998.

[20] A. Lakshman, P. Malik, and K. Ranganathan. Cassandra: A structured storage system on a P2P network. In *SIGMOD*, 2008.

[21] P. Sanders, S. Egner, and J. Korst. Fast concurrent access to parallel disks. In *SODA*, 2000.

[22] A. Schrijver. Bipartite edge-colouring in $O(\Delta m)$ time. *SIAM Journal on Computing*, 28(3):323–356, 1999.

[23] M. Stonebraker et al. C-Store: A column-oriented DBMS. In *VLDB*, 2005.

[24] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, February-April 2005.

[25] X. Zhou and T. Nishizeki. Edge-coloring and f-coloring for various classes of graphs. *Journal of Graph Algorithms and Applications*, 3(1):1–18, 1999.