# LinkBench: a Database Benchmark based on the Facebook Social Graph

Timothy G. Armstrong*[1], Vamsi Ponnekanti[2], Dhruba Borthakur[2], and Mark Callaghan[2]

[1]Department of Computer Science, University of Chicago

[2]Database Engineering Team, Facebook, Inc.

## ABSTRACT

Database benchmarks are an important tool for database researchers and practitioners that ease the process of making informed comparisons between different database hardware, software and configurations. Large scale web services such as social networks are a major and growing database application area, but currently there are few benchmarks that accurately model web service workloads.

In this paper we present a new synthetic benchmark called LinkBench. LinkBench is based on traces from production databases that store "social graph" data at Facebook, a major social network. We characterize the data and query workload in many dimensions, and use the insights gained to construct a realistic synthetic benchmark. LinkBench provides a realistic and challenging test for persistent storage of social and web service data, filling a gap in the available tools for researchers, developers and administrators.

## 1. INTRODUCTION

Much of the data powering Facebook is represented as a *social graph*, comprised of people, pages, groups, user-generated content and other entities interconnected by edges representing relationships. Such graph data models have become popular as sophisticated social web services proliferate.

At Facebook, persistent storage for the social graph is provided by an array of MySQL[1] databases called User Databases (UDBs). Facebook's memcached and TAO cache clusters [2, 3] cache large amounts of UDB data in memory. The vast majority of read requests hit in the cache, so the UDBs receive a production workload comprised of reads that miss the caches and all writes.

The Database Enginering team at Facebook has a growing need for benchmarks that reflect this database workload to assist with development, testing and evaluation of alternative database technologies. Facebook's software architecture abstracts storage backends for social graph data, allowing alternative database systems to be used.

Given the scale of Facebook's infrastructure, any changes in technology require careful evaluation and testing. For example, in the past we performed a thorough evaluation of HBase [4] as an alternative social graph storage backend. Facebook already uses HBase for major applications including its messaging backend [5]. Its benefits include easier load balancing, failover and strong random write performance. To accurately compare performance of MySQL and HBase, we mirrored part of the production workload on "shadow" clusters running our tuned and enhanced versions of MySQL and HBase. Contrary to expectation, MySQL slightly outperformed HBase in latency and throughput while using a fraction of the CPU and I/O capacity. Further experiment details are in Appendix A

The UBase benchmark effort took a lot of time and hard work to run and we want a more efficient way to evaluate alternative database solutions. Assessing new database systems will be crucial in the near future due to hardware trends such as solid state storage and increasing core counts. These new technologies could allow impressive performance gains, but systems well-suited to the previous bottleneck of rotating disks cannot always exploit the I/O capacity of solid state drives, or the processing power of many-core. Significant efforts are underway in industry and academia to better exploit this new generation of hardware, including key-value stores such as FlashStore [6], SILT [7], embedded databases such as WiredTiger [8] and new storage engines for relational databases such as TokuDB [9].

Many of these developments are promising, as are ongoing efforts to improve and adapt current technologies such as MySQL/InnoDB. We intend to conduct ongoing benchmarking on new and existing systems to guide decisions about the future of social graph storage at Facebook. Part of this plan involves development of a database benchmark that closely approximates our production workloads

The contributions of this paper are:

- A detailed characterization of the Facebook social graph workload using traces collected from production databases.
- LinkBench, an open-source database benchmark[1] that is closely based on our production workloads, and customizable for variant and alternative workloads.

*Work done while at Facebook


*SIGMOD* '12 New York, New York, USA


[1]To be made available at `https://github.com/facebook/`

## 2. RELATED WORK

There are a number of existing widely accepted benchmarks for database systems, some of which have been developed in industry and some in academia.

Mainstream database benchmarks do not closely match the Facebook UDB workload, so are of limited use. Transactional benchmarks such as TPC-C [10] are typically based on business transaction-processing workloads and extensively exercise the transaction handling properties required to maintain ACID. Analytic benchmarks such as TPC-H [11] focus on complex queries that involve large table scans, joins and aggregation. The Facebook UDB workload places different demands on database systems from either type of benchmark. Queries are fairly simple and short-lived with no full table scans or joins. While some ACID properties are required, transactions are simple and short-lived: the workload could be served by non-SQL database systems without general-purpose transactions.

Existing and proposed graph [12, 13] and object database benchmarks [14] operate on graph-structured data, not dissimilar to the social graph. These benchmarks have a significant focus on multi-hop traversals and other complex graph analysis operations, unlike the simpler retrieval operations that comprise the bulk of the Facebook workload.

The Yahoo Cloud Services Benchmark [15] is a benchmark designed to measure performance of different database systems, particularly distributed "cloud" database systems. The Facebook workload has similarities to the YSCB benchmark, but the data, supported operations, and workload mix are different. LinkBench's workload and data model is also grounded directly in measurements from production systems to increase our confidence in the relevance of the benchmark. Additionally, we are specifically interested in performance of the persistent storage layer because we handle row-level caching externally to the database. This focus simplifies understanding performance properties of design choices such as disk storage layout, without confounding factors arising from other design and implementation choices in a cloud database's caching and replication design.

## 3. WORKLOAD CHARACTERIZATION

We aim to construct a synthetic benchmark which can predict with reasonable accuracy the performance of a data storage layer for the Facebook production workload, and more generally for other web services that use social or graph-structured data. A synthetic benchmark has advantages in comparison to alternative approaches, such as capturing and replaying traces. We can share the benchmark with the broader research community without any risk of compromising users' privacy. It also allows the benchmark to be parameterized, allowing the simulated data and workload to be varied to test systems of different scales, and to explore different scenarios and workloads.

This section presents a characterization of Facebook's social graph workload, identifying key characteristics that we will replicate in a synthetic benchmark.

### 3.1 Social Graph Data Model

The social graph at Facebook comprises many *objects*, the nodes in the graph, and *associations*, directed edges in the graph. There are many different types of objects and associations. Examples of entities represented as objects include status updates, photo albums, or photos/video metadata:

| | | |
|---|---|---|
| id | int64 | unique identifier |
| type | int32 | type of object |
| version | int64 | tracks version of object |
| update_time | int32 | last modification (UNIX timestamp) |
| data | text | data payload |

(a) Object (graph node). id is unique key.

| | | |
|---|---|---|
| id1, id2 | int64 | ids of edge's endpoints |
| atype | int64 | type of the association |
| visibility | int8 | visibility mode of the edge |
| timestamp | int32 | a client-provided sort key |
| data | varchar | small additional data payload |

(b) Association (graph edge). (id1, atype, id2) is unique key.

Table 1: Database schema for social graph storage.

typically entities which have some associated data. Associations are a lightweight way to represent relationships between objects, for example if a user posted a photo, a user liked a photo, or if a user is friends with another user.

Table 1 shows the schema used to represent objects and associations. The *data* fields are stored as a binary string. A system at a higher level in the software stack supports richer data types with per-object-type schemas which can then be serialized into the data field. The *version* and *update_time* fields of the objects are updated with each change to the object's data, with the version incremented and update_time set to the current time. The *timestamp* field of an association is a general-purpose user-defined sort key (often a true timestamp), where high values are retrieved first. The *visibility* field allows data to be hidden without permanently deleting it, for example to allow a user to temporarily disable their account. Only visible associations are included in any query results (including counts).

### 3.2 Sharding and Replication

The entire Facebook graph is far too large to fit on a single server, so must be split into many *shards*. The nodes (objects) in the graph are allocated to shards based on *id*, with a function mapping the id to a shard. Associations (edges) are assigned to shards by applying the same mapping to *id1*, meaning that all out-edges for a node are in the same shard. Client applications have some control over the location of newly created objects. For example, a new object could be colocated with a related object (yielding some performance benefit from locality), or assigned to a random shard. The number of shards is chosen so that there are many shards per database instance, allowing rebalancing if necessary.

Each database instance has multiple replicas, with one master and multiple slaves. Replicas are geographically distributed, with reads handled from local replicas, which reduces latency and inter-datacenter traffic. Maintaining multiple replicas also allows for manual failover in the event of node or datacenter outages. All writes are applied synchronously at the master database, and replicated to slaves asynchronously (but under normal circumstances, quickly). In the current MySQL/InnoDB system, the data storage layer supports ACID, so the master MySQL instance has a fully consistent snapshot at all times. The overall system therefore provides timeline consistency [16], which is stronger than the eventual consistency supported by some other systems.

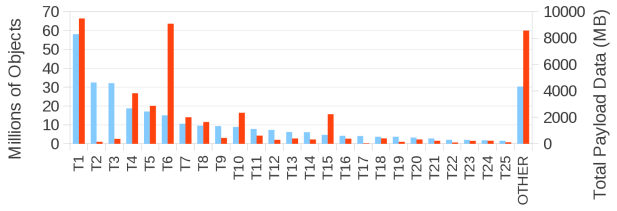
## 3.3 Structure of the Social Graph

One component of a benchmark is a data generator that can produce synthetic data with similar characteristics to real data. To understand the salient characteristics of our real data, such as graph structure and typical record size, we looked in detail at the data in a single MySQL instance.

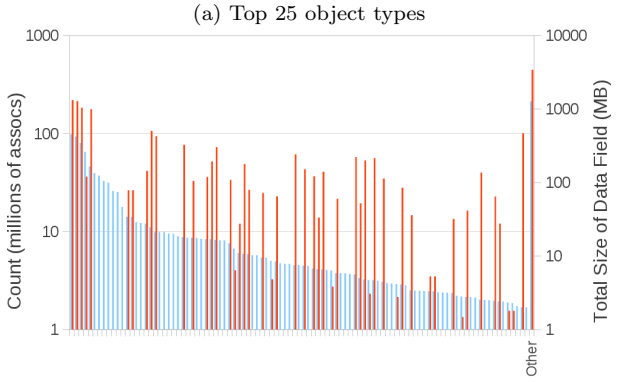
Due to some non-uniformity in the data between different shards, the numbers and charts presented in this section do not cover the entire data set, but we expect that they are representative.

### 3.3.1 *Object and Association Types*

Figure 1 shows the breakdown of graph data by object and association type. For both objects and associations we see that there are a few especially numerous types (where there are many instances of the type per user), but the tail of other types makes up a large part of the social graph.




(a) Top 25 object types



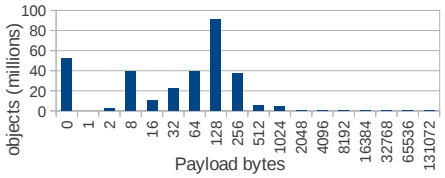

(b) Top 100 association types

Figure 1: Top social graph data types ranked by count. Count is light blue and payload size in dark red. A small number of object (graph node) and association (graph edge) types are far more common than the rest, with many instances per user. The "tail" of less common types comprises a large portion of the social graph.

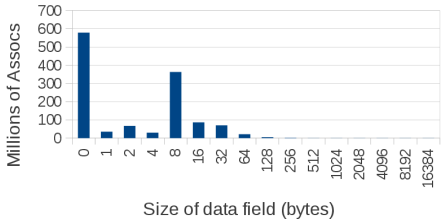
### 3.3.2 *Payload Data*

The mean payload per object is 87.6 bytes, while the average payload per association is much smaller at 11.3 bytes. 49% of associations have no payload. Figure 2 shows the overall distributions of data size for objects and associations. The distributions are similar to log-normal distributions, aside from the significant number of objects and associations with zero-size payloads. Figure 3 shows similar distributions for individual object types. This is also true for associations (not shown due to space limitations), although many association types always have no payload.

Payload data uses a mix of formats, including text-based and binary formats. Large payloads are compressed above the database tier. Compressibility of data affects system performance and file size because we use InnoDB compression to improve storage efficiency. Compressibility was estimated by sampling data from several sources and compressing with bzip2, an effective but slow compression algorithm. Results are shown in Table 2. Compressibility varies between sources, but 60% for objects and 30% for association payload data are representative compression ratios.




(a) Object payload size distribution




(b) Association payload size distribution

Figure 2: Payload size distributions. Both follow roughly log-normal distributions, aside from an additional peak at 0 bytes. In both cases payload sizes cluster within an order of magnitude around different modes. Histogram buckets are labeled with the lower bound.

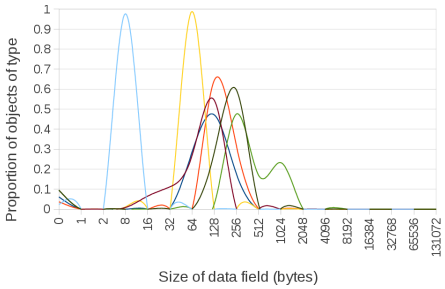



Figure 3: Size histograms for selected object types, illustrating that size distributions for different types shows roughly similar patterns with different peaks

| | Compression ratio |
|---|---|
| Objects in database | 61.3% |
| Object insert queries from trace | 46.0% |
| Object update queries from trace | 67.2% |
| Associations in database | 30.6% |
| Association insert queries from trace | 30.3% |
| Association update queries from trace | 57.5% |

Table 2: Compressibility of object payload data from different sources. Payload data from a random sample of rows was concatenated into a single file separated by newlines, and compressed using bzip2.

### 3.3.3 *Graph Structure*

Understanding certain properties of structure of the social graph is important in order to generate realistic benchmark data.

The outdegree distribution for each object is one important property. Every object has at least one out-edge, however there are also out-edges for ids that do not correspond to objects: some data types are allocated identifiers but are not represented as objects. Figure 4 shows the outdegree distribution. We see that the trend is consistent with a pareto distribution, but with a bulge showing more nodes with outdegree between 100 and 100,000 than a pareto distribution.

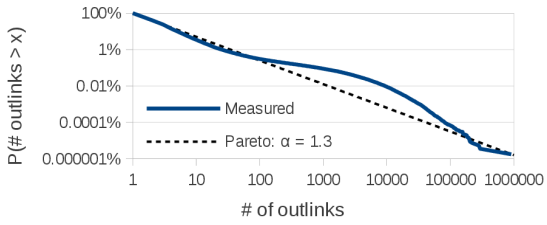



Figure 4: Distribution of node outdegree with logarithmic scale.

Previous analysis of online social networks [17, 18] has reported similar heavy-tailed power-law distributions for social networks comprised solely of people and relationships between people. Our results show that a power-law distribution still occurs with the additional variety of nodes and edges of the full Facebook social graph. We did not analyze the indegree distribution, as the sharding scheme means that doing so would have required accessing all database shards. However, we expect indegree to be correlated with outdegree, as many edge types in the social graph are symmetric, and past work [17] has reported strong correlation between indegree and outdegree in social networks.

There are many further properties of the social network graph structure that could be examined, such as clustering coefficient [19] or community structure [20]. However, for our purpose of modeling our database workload, it suffices to assume that there is no higher order graph structure in communities or correlations between properties of neighboring nodes. We believe that, for the operations in our workload, locality of access for a given id, and the average result size for range scans capture the main factors influencing performance of each operation in isolation. There are certainly patterns of locality in our social network workload as pages are generated for users and users navigate between related entities in the social graph. The amount of locality however, is likely to be relatively small and unpredictable, since aggressive caching outside of the database will absorb much of the locality. Since we are not modelling locality, the structure of the graph becomes less important for database performance and a generative model that neglects higher order graph structure is sufficient to measure performance of the storage layer.

## 3.4 Operation Mix

The set of operations used by the web front end and other services to access the social graph include standard insert, update and delete operations to modify data, and variations on key lookup, range and count queries. The set of operations covers most of the common access patterns required to serve data to users and is deliberately kept simple, to allow easier caching and system implementation[2].

[2]There are features, such as complex search queries, that cannot be efficiently implemented using this interface, but in practice are better provided by specialized services.

| Graph Operation | Result | # Queries |
|---|---|---|
| obj_get(ot, id) | object | 45.3M (12.9%) |
| obj_insert(ot, version, time, data) | id | 9.0M (2.6%) |
| obj_delete(ot, id) | - | 3.5M (1.0%) |
| obj_update(ot, id, version, time, data) | - | 25.8M (7.4%) |
| assoc_count(at, id) | count | 17.1M (4.9%) |
| assoc_range(at, id1, max_time, limit) | assocs | 177.7M (50.7%) |
| assoc_multiget(at, id1, id2_list) | assocs | 1.8M (0.5%) |
| assoc_insert(at, id1, id2, vis, time, version, data) | - | 31.5M (9.0%) |
| assoc_delete(atype, id1, id2) | - | 10.5M (3.0%) |
| assoc_update(atype, id1, id2, vis, time, version, data) | - | 28.1M (8.0%) |

Table 3: The set of social graph operations received by database instance over a six day period. Operation parameters and return values are shown, *ot* stands for object type and *at* stands for association type. Other parameters correspond to fields described in Section 3.1.

The queries issued to databases are classified into a few basic operations, shown in Table 3. These include:

- Point reads for associations and objects identified by primary key, with the option of batching multiple association reads batched into a single query.
- Simple create, delete and update operations for associations and objects identified by primary key.
- Association range queries for a given id and type and a timestamp range, ordered from latest to oldest. For example, a range query might obtain the node ids for the most recent comment on a post. A row limit, $N$, must be specified. The most recent $N$ associations before the provided timestamp are returned.
- Association count queries, for the number of visible out-edges of a given type from a given node. For example, a count query might count a user's friends.

To understand the database workload, we collected six days of traces of all social graph database queries issued by TAO, the distributed in-memory caching system through which Facebook's production web infrastructure accesses the social graph. We present in this section an analysis of the high-level trends and patterns.

Table 3 shows a 2.19 : 1 ratio of read to write queries, and a 3.19 : 1 ratios of association of object queries, with association range queries alone making up half of the workload. System load varies over time (see Figure 5) with a high base level and major and minor peaks every day.

| # Limit | % of range queries |
|---|---|
| 1 | 0.86% |
| 1000 | 0.39% |
| 6000 | 7.44% |
| 10000 | 91.54% |
| Other | 0.07% |

Table 4: Row limit for range queries observed in read workload. These statistics are determined by configuration of TAO cache clusters, rather than inherent in the workload.

Although the workload was fairly balanced between read and write operations, we saw 40.8 rows read per row written,

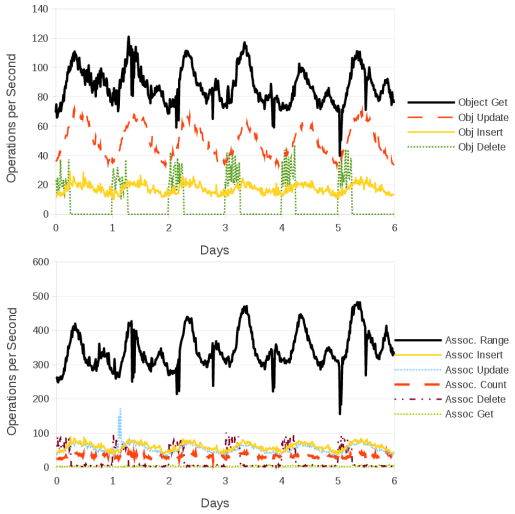



Figure 5: Temporal variation in number of operations in workload over time. Top: object operations. Bottom: association operations.

| # rows | % of range queries |
|---|---|
| 0 | 26.6% |
| 1 | 45.4% |
| 2 | 5.4% |
| 3-5 | 6.4% |
| 6-10 | 4.2% |
| 11-20 | 4.2% |
| 21-50 | 3.5% |
| 51-100 | 1.6% |
| 101-500 | 2.0% |
| 501-1000 | 0.4% |
| 1001-10000 | 0.3% |
| >10000 | 0.01% |

(a) Range scan row count distribution

| # keys | % of assoc. get queries |
|---|---|
| 1 | 64.6% |
| 2 | 8.8% |
| 3 | 3.1% |
| 4 | 1.9% |
| 5 | 1.6% |
| 6 | 1.4% |
| 7 | 10.6% |
| 8 | 0.8% |
| 9 | 7.1% |
| 10 | 0.2% |

(b) Lookup key count for multi-get queries.

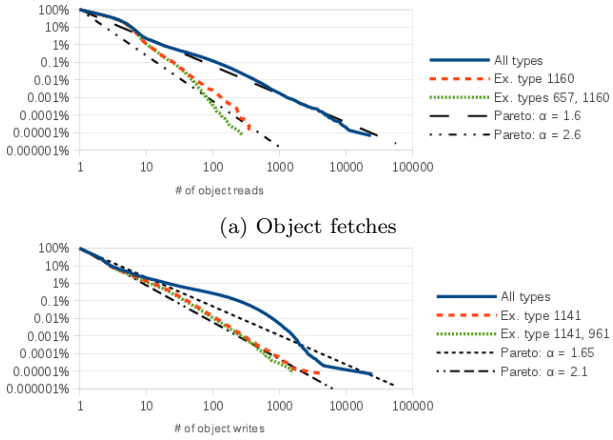
Table 5: Rows read for social graph edge read queries.

since write operations only affect a single row but many read operations return multiple rows. The workload is surprisingly read heavy given that all writes hit the databases, but cache clusters intercept most reads. Most range scans had a large upper limit on the result size, shown in Table 4. The large limits are due to aggressive caching that prefetches and caches ranges of associations. Analysis of read logs showed that the mean range scan result, ignoring limits, was 21.9 rows. Range limits only slightly reduce this: uniform limits of of 10000 or 6000 would result in 20.9 or 20.1 rows respectively. The average number of keys per association get query was 2.62. Table 5 shows distributions for both.

Most range queries were for the $n$ most recent items: 0.96% of range scan queries specified a maximum timestamp, typically because they were trying to fetch older history that was not retrieved in the first query.

## 3.5 Access Patterns and Distributions

In database deployments, some "hot" data is far more frequently accessed than other data, while there is often also "cold" data that is infrequently accessed, if at all. Stated differently, some rows of data are orders of magnitude more likely to be read or written than others. When constructing a synthetic benchmark, it is important to have realistic data access patterns because the interaction between patterns of hot and cold rows and a database system's caches is an important factor in overall performance.

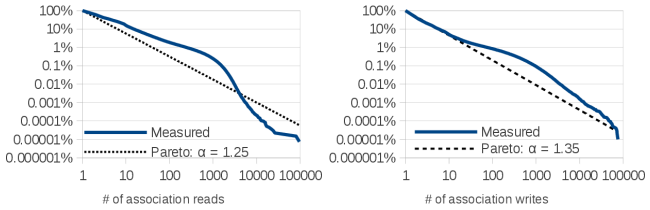



(a) Object fetches




(b) Object updates

Figure 6: Distribution of accesses for different kinds of operations based on id of object. Pareto distributions are shown for comparison. Distributions are also shown excluding the two top types for reads and writes, which exhibited unusual behavior.

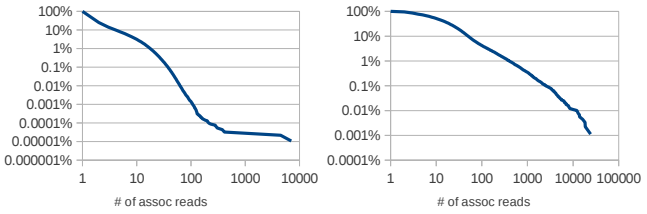



(a) association reads (get, count and range queries) (b) association writes (update, insert and delete queries)

Figure 7: Distribution of accesses for different kinds of operations based on id1/type of association.

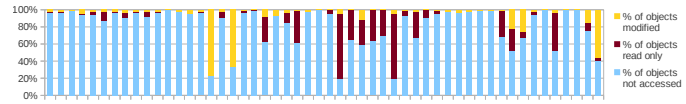
In order to do an initial characterization of the distribution of "hotness" for data, we examined the distribution of operations between different node ids. We aggregated all different varieties of reads and writes. Different categories of operation show similar skewed access distributions, where the majority of items are rarely accessed, but a small minority are read and written frequently. These patterns occur even in spite of the extensive caching of data outside of the database. Figure 6 and Figure 7 show the access distributions for objects and associations respectively. A power-law distribution, such as the pareto distribution, looks to be a reasonable approximation. We examined several of the most popular association and object types and determined that the power law pattern remains when looking at individual data types. Figure 8 shows the access distribution for the *like* association, one of the top association types.

In order to produce a realistic benchmark, we want to understand more accurately the causes of access patterns. We looked into a number of possibilities to better inform the design of LinkBench:

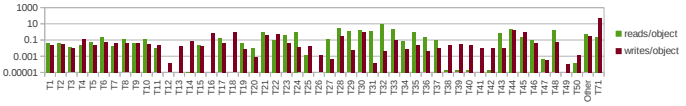



(a) Edges from user to object liked (b) Edges from objects to user they were liked by

Figure 8: Distribution of reads for the "like" association showing that power law behavior is still present for individual association types.




(a) Cold and read-only data by type, showing only 4.78% of objects were modified and only 8.73% were accessed in six day period.




(b) Intensity of reads and writes per row of data.

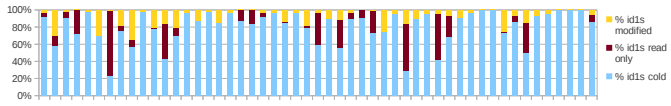
Figure 10: Workload metrics for top object types, illustrating the disparity in data access patterns for different types.

- Some of the variation may be explained by different access patterns for different types. For example, a person's profile will be more frequently accessed than any particular post. We discuss this in Section 3.6.
- The "popularity" of a node in the graph: for example, a post that has been shared by many people, or a page for a popular figure with many followers will be more frequently accessed. Objects with high degrees are more "discoverable" and more likely to be read, due to the large number of path through the graph that lead to them. They may also accumulate more new edges, due to processes such as preferential attachment that are commonly believed to occur in social networks[21], where nodes in a graph with high degrees are more likely to accumulate more connections to other edges as the graph evolves. We explore this possibility in Section 3.6.

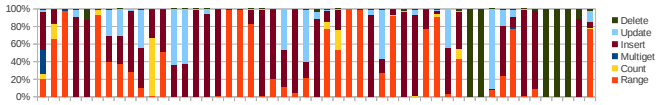
## 3.6 Access Patterns by Data Type

Figure 10 illustrates varying access patterns for different object types, with different types having widely varying ratios of reads to writes. We looked at what fraction of objects were never accessed, or *cold*. Overall a large proportion of objects, 91.3%, are cold data that was never accessed during the 6 day trace. 95.2% were read-only during that period. Some types are far more intensely read and written than other types, with average read and write intensity varying by two to three orders of magnitude between types.

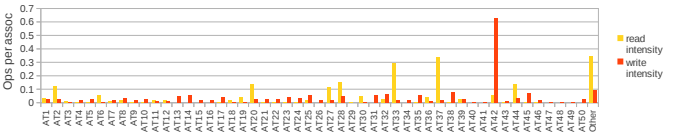
Access patterns for associations are more complex, because of the variety of supported operations, and because range queries return variable numbers of rows. Figure 11 compares metrics between association types. As with objects, the workload varies greatly between association types in composition of queries and frequency of reads and writes.




(a) Proportion of association lists (identified by unique id1, assoc_type pair), which are accessed or modified. Overall 3.2% of lists were modified and 7.8% were accessed.




(b) Relative proportions of different operations per association type, illustrating the widely different workloads.




(c) The number of operations relative to the total number of associations of that type.

Figure 11: Workload metrics for top association types, illustrating the disparity in data access patterns.

We looked at what proportion of associations were cold. Breaking down the associations into lists, identified by a (id1, assoc_type) pair, we saw that 92.2% of these lists were cold and not the subject of any read or write operations in 6 day period and 96.6% were not modified. 23.3% of queried lists were only counted, without any association data returned. Interestingly, 92.2% of association lists were cold, but only 74% of association rows were cold. This indicates a correlation between the length of an association list and the likelihood of it being accessed.

## 3.7 Graph Structure and Access Patterns

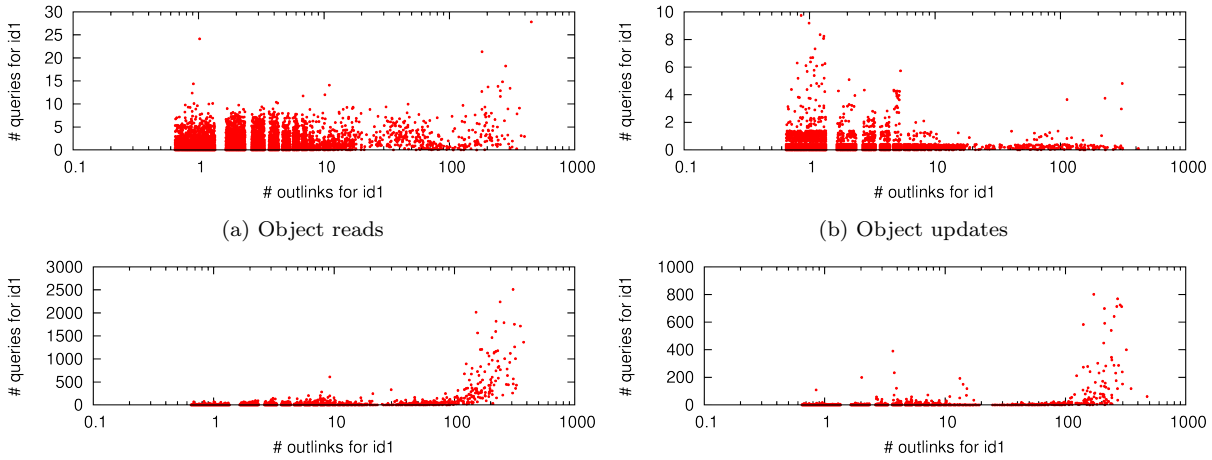
One important question is whether there is a relationship between the structure of the graph and the frequency of access of different nodes and edges in the graph. To investigate, we took a random sample of 1% of the ids with at least one out-edge in the database, and compared the outdegree with the number of queries for that id in the trace. Figure 9 shows the results for various classes of queries. There is a correlation between outdegree and association read queries (mostly range scans), while there is little correlation for node read queries, possible because simple object retrievals are cached more effectively than complex association queries. Similar patterns can be seen for write queries. This indicates that a realistic benchmark needs to have a query mix that is biased towards graph nodes with high outdegree.

## 3.8 Update characterization

The nature of in-place updates may have some impact on performance of the system, for example causing fragmentation if data shrinks, or forcing additional page allocations.

Updates to objects always update the version, timestamp and data fields. Updates to associations often only update one or two fields, such as the timestamp or the visibility, as shown in Table 6.

Typically the payload data size only changes by a small amount, illustrated by Figure 12. For objects, over a third




(a) Object reads

(b) Object updates

(c) Association read queries

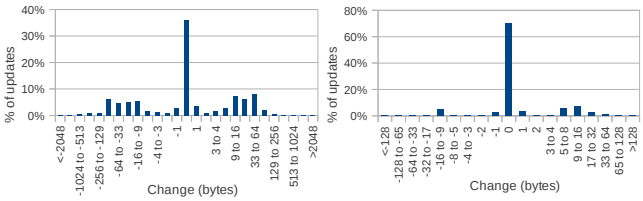
(d) Association insert/delete/update queries

Figure 9: Correlation between social graph node's outdegree and read/write frequency. The outdegree is correlated with operations on edges, but not operations on nodes. Jitter added to show density.

| Field | % Assoc. Updates |
|---|---|
| Visibility | 12.0% |
| Timestamp | 84.4% |
| Version | 98.4% |
| Data | 46.3% |

Table 6: Fields modified by association update operations.

of updates do not change the data size, while the majority of other updates alter it less than 128 bytes. Associations exhibit a similar pattern. In both cases, when the data size stays constant it is typically because a data field, such as a number or other identifier, is modified in such a way that the length of the representation does not change.




(a) Object updates

(b) Association updates

Figure 12: Change in payload data size for update operations. Most data updates only change the size a small amount.

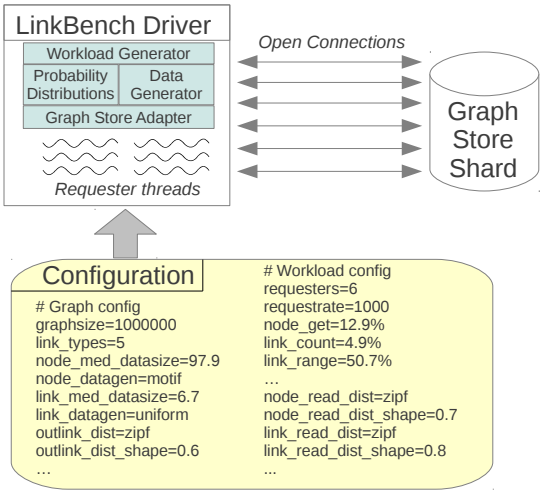



Figure 13: LinkBench architecture, showing the datastore under benchmark (which could logically be considered a single shard of a larger system), a subset of configuration settings, and the internal components of the LinkBench driver.

## 4. BENCHMARK DESIGN

In this section we present the LinkBench database benchmark, describing the architecture of the system, the configurable building blocks that allow the benchmark to be customized, and the process of generating a synthetic social graph and database workload of graph operations.

The benchmark is designed to test performance of a single database instance in isolation. We have a client-server architecture, shown in Figure 13 with the Linkbench client implemented in Java driving a *graph store*. We currently have implemented a MySQL graph store, but any database system that meets the requirements in Section 4.1 can be benchmarked. We describe key decisions made in the design of the client in Section 4.2.

The LinkBench driver operates in two phases. The first phase populates the graph store by generating and bulk-loading a synthetic social graph. The generative model used is described in Section 4.3 and Section 4.4. In the second phase the driver benchmarks the graph store with a generated workload of database queries and collects performance statistics. Generation of the simulated workload is discussed in Section 4.5 and the metrics collected are discussed in Section 4.6. Both phases have many configurable parameters that can be used to scale up or down the workload, or to explore workloads with different characteristics.

## 4.1 LinkBench Graph Store Implementation

LinkBench is designed so that the same benchmark implementation can be used for many different database systems. A database can be used as a LinkBench *graph store* with an adapter implementing the operations in Table 3.

To ensure comparability of benchmark results, we impose some constraints on the implementation. The entire social graph should be stored in persistent storage. Any database schemas, compression, indices or other configuration or optimization should be reported. All writes must be durable, with data flushed to persistent storage before the operation completes so that the data can be recovered in the event of a crash. Any update operations should be atomic and ensure consistency. For example, in our MySQL benchmark implementation, a separate edge count is updated in the same atomic transaction as an edge insertion or deletion. Any weaker ACID properties should be disclosed.

## 4.2 Tradeoffs in Design of Benchmark Client

In designing the benchmark we kept a balance between realism and simplicity. The preceding analysis of the workload indicates there are many complex patterns, correlations and internal variations in our database workload, many of which have implications for database design and performance. It is important for us that the benchmark be a reasonably good predictor of the real-world performance of a database system, so we must capture some of these features in the benchmark. However, in some cases we did not try to fully replicate complex patterns, where it would have required excessive memory or CPU in the client, where it would have perturbed performance of the system under test, or led to an excessively complex benchmark with results difficult to understand or replicate.

Some important patterns in the workload are data-dependent, where the structure of the graph affects the queries issued by the client, for example where nodes with high outdegree frequently queried. It is not feasible to track this data in memory for a large database, which presents a challenge in creating a scalable and realistic benchmark.

LinkBench uses multiple concurrent requesting threads in order to allow a high throughput of queries and to test performance of concurrent queries. Sharing state between these threads complicates scaling and implementation of the benchmark client, and also complicates horizontal scaling of the benchmark client across multiple nodes if needed.

For these reasons, we decided to keep the workload generator stateless with a couple of exceptions, only using configuration parameters and knowledge about how the graph was generated to make decisions about the next operation to execute. The statelessness contrasts with real-world clients, which are stateful: caches cause some temporal anti-locality, while navigation patterns of users induce some spatial locality, with bursts of activity in sections of the graph. We believe that the locality effects will be limited and unpredictable due to the aggressive caching, so we do not emulate them.

## 4.3 Workload Generator Building Blocks

LinkBench uses a range of configurable and extensible building blocks so that the benchmark can be tweaked and customized. The benchmark configuration file contains many modifiable parameters, and allows different implementations of these building blocks for graph creation and workload generation to be swapped in.

LinkBench has a framework for *probability distributions*, which are used in many places in the benchmark to generate random data. Distributions are implemented as Java classes and include the uniform distribution, the Zipf distribution, and the log-normal distribution. Wherever a distribution is used in LinkBench, the implementation and parameters can be configured. A distribution provides two functions: a quantile function that allows, for example, calculation of outdegree of the graph node with the $k$th highest outdegree out of $n$ nodes; and a choose function that selects integers in a range $[1, n]$ with probability weighted by the distribution.

The weighting works such that the lowest keys are most popular, meaning that popular database rows would be clustered together if the values are used directly as database row keys. In real data sets, popular data is scattered throughput the key space. Other benchmarks shuffle popular data throughout the key space by permuting each key $i$ within the range of valid keys $[1, n]$ using a permutation function $p(i)$. Gray suggests multiplying the index by a prime modulo $n$ to obtain the new key [22]. YCSB [15] generates keys within a much larger range, and shrinks the range by hashing.

In LinkBench, we want correlated distributions for access frequency and node outdegree, while generating data in bulk in primary key order. In order to achieve this $p^{-1}(i)$, the inverse of the prior permutation, needs to be efficiently computable. Both permutation functions mentioned previously are difficult to invert, so LinkBench uses a different, invertible, *permutation function*. It can be given different parameters to alter the permutation, and has low CPU and memory overhead. If there are $n$ items in the keyspace, we choose a number $k$, for example $k = 1024$. We then fill an array $A$ with $k$ pseudorandom integers (using a known seed for reproducibility). If $n$ is divisible by $k$, then the permutation is computed as $p(i) = ((i + k \cdot A[i \bmod k]) \bmod n)$, which rotates each set of indices with the same remainder modulo $k$ in the keyspace. The inverse is easily computable using the same formula with $A[i \bmod k]$ negated. For Linkbench, we generalized the formula for the case where $n$ is not divisible by $k$. This method of permuting data can key distribution sufficiently with limited memory overhead.

LinkBench also has a framework for *data generators*, which can fill byte buffers with randomly generated data, useful when adding payload data for graph nodes and edges. The main one used in LinkBench is the *motif data generator*, which generates a configurable mix of random bytes and repeated multi-byte motifs.

## 4.4 Generative Model for Social Graph

In this section we describe the generative model used to construct a social graph. Generating random graphs with structure close to real social networks is challenging and an active area of research. For the purposes of the benchmark, we do not need full fidelity to the original graph structure. Rather, we want a simple, configurable and fast graph generator give results close to the real social graph in the right dimension so that it places similar stresses on the database. The degree distribution of the generated data must be realistic, so that similar numbers of records are scanned by range queries. However, the community structure of the generated graph (e.g. the probability of two friends having another mutual friend) is unimportant, as this does not directly affect the performance of any queries in the workload.

#### 4.4.1 Graph Size

LinkBench can be run with different graph sizes by specifying the *initial node id range*. For example, if a range of $[1, 1000001]$ is specified, then 1000000 nodes and corresponding edges will be bulk loaded. The graph will continue to expand in the later benchmark phase.

For full-scale benchmarking we use graphs with around 1 billion nodes occupying approximately 1TB using InnoDB without compression. A social graph of this size can be generated and loaded by LinkBench in around 12 hours on a high-end servers with solid state drives thanks to bulk-loading optimizations such as batch insertions.

#### 4.4.2 Generating Graph Nodes

The simpler part of generating a synthetic social graph is generating graph nodes (also referred to as objects). We have simplified the benchmark by only having a single node type in the graph. The major downside of this is that we cannot have different types of nodes with different workload characteristics. The simplification of the benchmark implementation is considerable, as without this simplification, to select a random node ID of a given type to query would require the benchmark client to track which parts of the id space are of which type. This is challenging and memory-intensive when new IDs are being allocated during the benchmark. This is a good compromise, since node queries are a small portion of the workload compared to edge queries, and much of the variation in access patterns is captured by the access distributions used when generating node queries.

Node payload data is generated using the *motif generator* with parameters chosen to get a compression ratio of approximately 60%, similar to the measured compression ratio in Table 2. The size of the payload is chosen from a configured probability distribution. We use a log-normal distribution with a median of 128 bytes.

#### 4.4.3 Generating Graph Edges

Given the varying access patterns for different association types seen in Section 3.6, we explored the possibility of a benchmark that incorporated a range of distinct association types. However, in the end we decided against attempting to faithfully replicate the diversity of associations, mainly because we could not justify the additional complexity when it was possible to capture much variation with a homogenous model of edges. We support a configurable number of edge types, but all use the same data and workload generator.

Graph edges, which we term *links* in the benchmark, are generated concurrently with graph nodes during bulk loading. We divide the node id space into chunks based on the id of the source node. The chunks are processed in parallel to speed loading. The chunks are processed in approximate order of id, and within each chunk strictly in order of id, improving the locality of access and speeding up load times. Loading in primary key order leads to lower fragmentation for B-tree-based database systems as tree nodes are filled close to capacity. Over time, as rows are added and removed, fragmentation of the database will tend to increase, leading to increased storage usage and somewhat degraded performance as caches can fit fewer rows. Database benchmarkers should be aware of this phenomenon, particularly when examining storage efficiency.

For each node the steps to generate edges are:

- Choose the outdegree deterministically using a probability distribution and shuffler. We use the measured outdegree distribution from Section 3.3.3 directly.
- Divide the outlinks between the different link types in a round robin fashion, in such a way that the $i$th type always has at least as many edges as the $i+1$th type.
- Select the id of target nodes for the $j$th link of each link type to be source_id + $j$. This makes it simple to determine during later workload generation which graph edges are likely to exist.
- Generate payload data for each edge using the motif data generator with compression ratio of approximately 30%, in line with Table 2.

### 4.5 Generating workload

Our workload generator comprises many threads of execution, all of which execute the same randomized workload. Statistics are collected by each thread and then aggregated at the end of the benchmark.

#### 4.5.1 Node Selection

As discussed previously, some of the most important features of the benchmark workload are the access patterns for different data types and operations: the distribution of reads and writes between graph nodes and edges. In this section we discuss the approaches used to select the ids for nodes used for operations.

The access patterns for node reads, node writes, link reads and link writes are separately configurable using the previously described probability distribution framework. We use the algorithm described by Gray et al. [22] to implement a Zipf probability distribution that is used for node accesses, with parameters calculated based on the fitted pareto distributions in Section 3.5.

The most straightforward access patterns are for node queries, where only the node id must be chosen. Since we observed that node query frequency was uncorrelated with the number of out-edges, we use a different shuffler to that used to generate outdegree.

For edge queries, we saw a loose correlation between row hotness and outdegree in the real workload. In order to simulate this loose correlation two access distributions are combined: one with the same shuffler as the outdegree distribution, giving a perfect correlation between access frequency and outdegree, and another with a different shuffler and no correlation. The distributions are blended by selecting from the correlated distribution with probability $p_{\text{corr}}$ and the uncorrelated with probability $1 - p_{\text{corr}}$ $p_{\text{corr}}$ is selected such that the mean range scan size approximately matched the empirical observations.

Some link operations (multiget, add, delete, update) require that the id or ids of the target node for the link is provided. We observe that for some of these operations (delete, update) the link should be present, for others (add) the edge should be absent, and for others (multiget) a mix of hits and misses is desirable. It is not practical for the client to track which links exist for a large database, so we exploit the knowledge that edges from node $i$ to nodes $[i..i + \text{outdegree}(id) - 1]$ were in the initial graph to choose target node ids with a given probability of the edge existing. Edges are added or removed during benchmarking, so

to handle the cases of inserting existing edges and updating non-existing edges, a single combined insert/update operation that inserts if not present or updates if present is used.

#### 4.5.2 Arrival Rate

In order to generate latency/throughput curves, we want to be able to control the arrival rate of new operations. We assume that the average arrival rate (a configurable parameter) remains constant over the course of the benchmark and choose the interval between arrivals from an exponential distribution.

#### 4.5.3 Operation Mix

Given the timing of an operation, we then need to select the operation to execute and then the parameters of that operation. We do not attempt to capture any temporal correlation between different operation types. The steps for all operations are the same:

1. An operation from Table 3 is selected. The measurements of the operation mix in Section 3.4 are used to select which operation to execute.
2. The id (id of the node, or id1 of the link) is chosen as described previously
3. For link queries, a link type is selected uniformly.
4. For link point queries, the number of link is selected using a geometric distribution with $p = 0.382$, yielding the same mean as observed (2.615 ids per query).
5. Any required target node ids of links are chosen as described previously.
6. For data modification, node or link fields are filled in with the same method as in initial data generation phase. This is not a perfect approximation, since the workload characterization in Section 3.8 revealed that many updates only made a small change to data size, or did not change some fields at all. This inaccuracy may slightly hurt update performance on storage layers that perform in-place modification of data, since the additional churn in data size and values may result in more fragmentation, page splitting and dirty pages.
7. For link range queries, a fixed result limit of 10000 is used, which should be reasonably reflective of the real workload since that was the value for over 90% of queries observed, and very few queries return more than 1000 rows. By default, the queries return the most recent rows, but a small fraction are *history queries*, which may specify a maximum timestamp. The client generates history queries by maintaining a fixed-size cache of (id1, link_type, timestamp) records which are added whenever a range query returns 10000 rows (which indicates there is likely more history past the oldest timestamp). This simulates a process where a client, after retrieving the first 10000 entries in a list of links, may later retrieve further history.

## 4.6 Metrics

There are a number of key metrics that we want Linkbench to measure. The most important metrics for speed are operation latency and throughput. We measure latency in Linkbench from the time when the operation to be executed is selected in the Linkbench client, until the time when the client receives all result data for a read operation, or receives confirmation of completion for a write operation.

The mean operation throughput should be reported, along with the latency statistics for each operation type that are reported by LinkBench: latency at 50th, 75th, 95th, 99th percentiles, maximum latency and mean latency.

Latency versus throughput curves can be obtained by varying the arrival rate of operations. A complete comparison of two systems will show a complete curve. Latency for specific operation types at the given level of throughput can also be reported.

Price/performance is also important, so for comparison of commercial systems, peak throughput per dollar for the full system (hardware and software) should be reported.

Several measures of resource utilization by the database system under test should be collected at one second intervals:

- CPU usage: user, system, idle, and wait.
- Read and write I/O operations per second.
- Read and write I/O rate in MB/s.
- Resident memory size.
- Persistent storage size, including temporary indices, tables and logs.

All of these metrics are useful for understanding system performance and efficiency. Storage size has become increasingly important as the bottleneck for systems with solid state disks is often capacity rather than I/O.

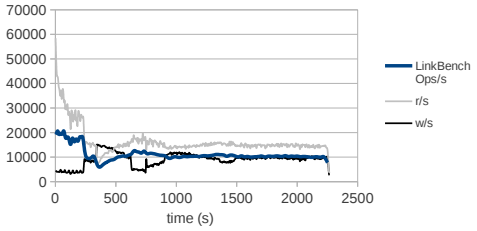
## 4.7 Validating Benchmark Configuration

Although LinkBench is customizable, we also focused on creating a workload configuration that would closely match the workload characterized earlier in this paper. This section summarizes how the match between LinkBench and our real workload can be validated in certain important dimensions.

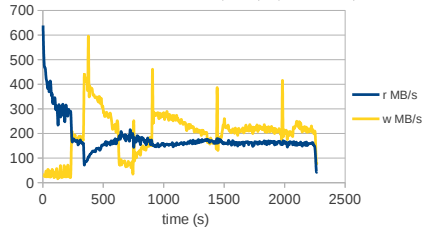
The generated graph matches in several dimensions by construction: the outdegree distribution exactly matches the empirical outdegree distribution, while node and link payload data sizes follow a similar log-normal distribution, and have the same compression ratios.

The workload generated also matches in several dimensions by construction. The mix of different operation types is the same, and the distributions of reads and writes to nodes follow power-law distribution with empirically derived exponents. The mean number of keys per multiget is the same, and has a similar skewed distribution.

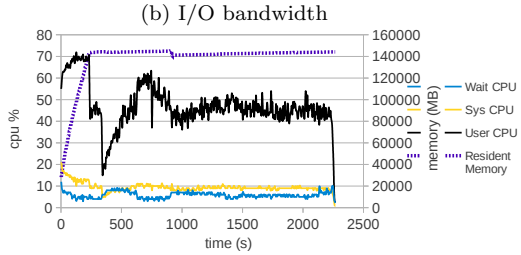
One important property of the workload that we could not guarantee by design was the mean number of result rows for range queries, which was measured at approximately 21 rows. Our first attempted configuration lead to an average result size of several hundred rows. We brought this down to $20 - 30$ rows by modifying the configuration in several ways. Links were split into two different link types, halving average length. We limited link history queries, which tend to have larger results, to 0.3% of range queries, less than the 0.96% observed. We finally set $p_{\text{corr}} = 0.005$ for link reads, so the outdegree-correlated distribution is used only 0.5% of the time.




(a) Benchmark throughput (ops/s), and I/O operations




(b) I/O bandwidth




(c) CPU and memory utilization

Figure 14: Operation throughput and system resource utilization over time.

| | mean | p25 | p50 | p75 | p95 | p99 | max |
|---|---|---|---|---|---|---|---|
| node_get | 1.6 | 0.4 | 0.6 | 1 | 9 | 13 | 191 |
| node_insert | 4.2 | 1 | 3 | 5 | 12 | 20 | 142 |
| node_delete | 5.2 | 2 | 3 | 6 | 14 | 21 | 142 |
| node_update | 5.3 | 2 | 3 | 6 | 14 | 21 | 143 |
| link_count | 1.3 | 0.3 | 0.5 | 0.9 | 8 | 12 | 65 |
| link_range | 2.4 | 0.7 | 1 | 1 | 10 | 15 | 2064 |
| link_multiget | 1.7 | 0.5 | 0.8 | 1 | 9 | 14 | 53 |
| link_insert | 10.4 | 4 | 7 | 14 | 25 | 38 | 554 |
| link_delete | 5.1 | 0.5 | 1 | 7 | 19 | 31 | 468 |
| link_update | 10.3 | 4 | 7 | 14 | 25 | 38 | 554 |

Table 7: MySQL LinkBench operation latencies in ms.

# 5. MYSQL BENCHMARK RESULTS

In this section we present results of benchmarking MySQL with LinkBench. The system under test is MySQL 5.1.53 with the Facebook patch. MySQL was configured with a 120GB InnoDB buffer pool, and the link table partitioned 32 ways to reduce mutex contention. Full durability was enabled with logs flushed to disk at transaction commit, and a binary log for replication generated. Separate hosts were used for the LinkBench client and MySQL server. The MySQL host had 2 CPU sockets, 8+ cores/socket, 144GB of RAM and solid-state storage with read latency at 16kB less than 500$\mu$s.

In order to ensure that benchmark performance was not bottlenecked by the LinkBench client, we did several runs while monitoring the client. The MySQL server was saturated using only a fraction of client CPU and network capacity. To double-check this result, we ran two LinkBench clients concurrently on different hosts and confirmed that this did not increase overall operation throughput.

A graph with 1.2 billion nodes and approximately 5 billion links was generated, which occupied 1.4TB on disk.

We performed a benchmark run with 50 concurrent requesting threads performing 25 million requests in total. Figure 14 shows benchmark throughput and resource utilization and Table 7 reports operation latencies at different percentiles. The benchmark took 2266 seconds, for an average throughput of 11029 requests a second. The system goes through a warm-up phase as the InnoDB buffer pool is populated with pages from disk and those pages are dirtied with writes. After a period it enters a steady-state phase. During the steady-state phase I/O read utilization remains high, indicating that the working set of the benchmark is larger than main memory. The high rates of I/O operations and I/O throughput highlight the benefit that MySQL/InnoDB can derive from solid-state storage.

# 6. CONCLUSION

We have presented the motivation and design of LinkBench, a database benchmark that reflects real-world database workloads for social applications.

We characterized the social graph data and accompanying database workload for Facebook's social network, extracting key statistical distributions and showing how power law distributions occur in several places.

We then described the design and construction of a benchmark that mimics the key aspects of the database workload, and presented a performance profile of the MySQL database system under this workload.

The benchmark software has been released as open source and we hope can be used by others to profile and experiment with other database systems. We will extend LinkBench with adapters for further database systems as we continue to evaluate new database technology for use at Facebook.

# 7. ACKNOWLEDGMENTS

Tien Nguyen Hoanh, an intern at Facebook in 2011, performed some initial data collection and development for LinkBench. Tim Callaghan at Tokutek provided valuable feedback on early versions of LinkBench. Scott Chen at Facebook helped with comparison of HBase and MySQL.

# 8. REFERENCES

[1] Oracle Corporation, "MySQL 5.6 reference manual," 2012, http://dev.mysql.com/doc/refman/5.6/en/.

[2] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. SIGMETRICS'12*. New York, NY, USA: ACM, 2012, pp. 53–64.

[3] Facebook, Inc., "TAO: Facebook's distributed data store for the social graph," 2012, draft in preparation.

[4] The Apache Software Foundation, "Apache HBase," 2012, http://hbase.apache.org.

[5] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer, "Apache Hadoop goes realtime at Facebook," in *Proc. SIGMOD'11*, 2011, pp. 1071–1080.

[6] B. Debnath, S. Sengupta, and J. Li, "FlashStore: high throughput persistent key-value store," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 1414–1425, Sep. 2010.

[7] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "SILT: a memory-efficient, high-performance key-value store," in *Proc. SOSP'11*. New York, NY, USA: ACM, 2011, pp. 1–13. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043558

[8] WiredTiger, Inc., "WiredTiger reference guide 1.3.4," 2012, http://source.wiredtiger.com/1.3.4/.

[9] TokuTek Inc., "TokuDB," 2012, http://www.tokutek.com/.

[10] Transaction Processing Performance Council, "TPC benchmark C standard spec. 5.11," Feb 2010, http://www.tpc.org/tpcc/spec/tpcc_current.pdf.

[11] ——, "TPC benchmark H (decision support) standard spec. 2.14.4," Apr 2012, http://www.tpc.org/tpch/spec/tpch2.14.4.pdf.

[12] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vañó, S. Gómez-Villamor, N. Martínez-Bazán, and J. Larriba-Pey, "Survey of graph database performance on the HPC Scalable Graph Analysis Benchmark," in *Web-Age Information Management*, H. Shen, J. Pei, M. Özsu, L. Zou, J. Lu, T.-W. Ling, G. Yu, Y. Zhuang, and J. Shao, Eds. Springer Berlin / Heidelberg, 2010, vol. 6185, pp. 37–48.

[13] D. Dominguez-Sal, N. Martinez-Bazan, V. Muntes-Mulero, P. Baleta, and J. Larriba-Pey, "A discussion on the design of graph database benchmarks," in *Performance Evaluation, Measurement and Characterization of Complex Systems*, R. Nambiar and M. Poess, Eds., 2011, vol. 6417, pp. 25–40.

[14] M. J. Carey, D. J. DeWitt, C. Kant, and J. F. Naughton, "A status report on the OO7 OODBMS benchmarking effort," in *Proc. OOPSLA'94*. New York, NY, USA: ACM, 1994, pp. 414–426.

[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. SoCC'10*. New York, NY, USA: ACM, 2010, pp. 143–154.

[16] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1277–1288, Aug. 2008.

[17] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proc. IMC'07*. New York, NY, USA: ACM, 2007, pp. 29–42.

[18] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow, "The anatomy of the Facebook social graph," *CoRR*, vol. abs/1111.4503, 2011.

[19] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks." *Nature*, vol. 393, no. 6684, pp. 440–442, Jun. 1998.

[20] M. Girvan and M. Newman, "Community structure in social and biological networks," *Proc. Nat'l Acad. Sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[21] M. Newman, "Power laws, Pareto distributions and Zipf's law," *Contemporary physics*, vol. 46, no. 5, pp. 323–351, 2005.

[22] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," *SIGMOD Rec.*, vol. 23, no. 2, pp. 243–252, May 1994.

# APPENDIX

## A. HBASE/MYSQL EXPERIMENT

The HBase/MySQL comparison was begun in 2011 with the goal of reducing total cost of storing massive amounts of User DataBase (UDB) data. At this time, HBase was already in production deployment for Facebook Messages, so was an obvious candidate. In addition by design it is a high-write-throughput database and maintains three synchronous local data replicas for quick local failover.

A MySQL and a HBase cluster were both set up to receive a portion of production requests. The HBase cluster had five machines: a dedicated HDFS NameNode, a dedicated HBase master and three nodes running both HDFS Datanode and HBase Region Server. Facebook's internal branches of HBase (roughly corresponding to HBase release 0.94) and HDFS were used. A native C++ client for HBase was developed and used for benchmarking. Data was compressed using the LZO compression algorithm. The MySQL cluster had three machines each running one MySQL server. Data was compresed until zlib. Both MySQL and HBase servers were set up so that 8GB of memory was usable for data caching (the OS buffer cache was disabled). In-house experts for both MySQL and HBase were involved in tuning and optimizing both systems. The benchmark process led to many HBase code enhancements to reduce latency and I/O usage.

We measured the 99th percentile latencies of several operations. Latencies were similar or markedly lower on MySQL.

| | MySQL p99 Latency | HBase p99 Latency |
|---|---|---|
| assoc_range | 25.3ms | 54.8ms |
| assoc_get | 21.9ms | 39.0ms |
| assoc_insert | 39.2ms | 56.3ms |
| assoc_delete | 49.9ms | 52.3ms |

System resource utilization was markedly different between MySQL and HBase processing the same workload. CPU utilization for HBase servers fluctuated between 20% and 35%, while it remained steady at around 5% for the MySQL servers. I/O operations per second varied greatly with HBase, varying sharply from 1000 up to above 2000, while MySQL consistently used 1200-1400.

This experiment showed that HBase consumed more CPU and incurred more I/O operations for the Facebook Graph workload. It also demonstrated the challenges in building custom tools to shadow production load onto test systems.