# Lookup Tables: Fine-Grained Partitioning for Distributed Databases

Aubrey L. Tatarowicz [#1], Carlo Curino [#2], Evan P. C. Jones [#3], Sam Madden [#4]

[#] *Massachusetts Institute of Technology, USA*

[1] altat@alum.mit.edu

[2] krl@yahoo-inc.com

[3] evanj@csail.mit.edu

[4] madden@csail.mit.edu

***Abstract*—The standard way to scale a distributed OLTP DBMS is to horizontally partition data across several nodes. Ideally, this results in each query/transaction being executed at just one node, to avoid the overhead of distribution and allow the system to scale by adding nodes. For some applications, simple strategies such as hashing on primary key provide this property. Unfortunately, for many applications, including social networking and order-fulfillment, simple partitioning schemes applied to many-to-many relationships create a large fraction of distributed queries/transactions. What is needed is a *fine-grained* partitioning, where related individual tuples (e.g., cliques of friends) are co-located together in the same partition.**

**Maintaining a fine-grained partitioning requires storing the location of each tuple. We call this metadata a *lookup table*. We present a design that efficiently stores very large tables and maintains them as the database is modified. We show they improve scalability for several difficult to partition database workloads, including Wikipedia, Twitter, and TPC-E. Our implementation provides 40% to 300% better throughput on these workloads than simple range or hash partitioning.**

## I. Introduction

Partitioning is an essential strategy for scaling database workloads. In order to be effective for web and OLTP workloads, a partitioning strategy should minimize the number of nodes involved in answering a query or transaction [1], thus limiting distribution overhead and enabling efficient scale-out.

In web and OLTP databases, the most common strategy is to horizontally partition the database using hash or range partitioning. This is used by commercially available distributed databases and it works well in many simple applications. For example, in an email application, where each user only accesses his or her own data, hash partitioning by user id effectively places each user's data in separate partitions, requiring very few distributed queries or transactions.

In other cases, such as social networking workloads like Twitter and Facebook, there is no obvious partitioning that will result in most queries operating on a single partition. For example, consider placing one user's posts in a single partition (partitioning by author). A common operation is listing a user's friends' recent posts, which now needs to query multiple partitions. Alternatively, replicating a user's posts on his or her friends' partitions allows this "recent posts" query to go to a single partition, but requires updates to be sent to multiple partitions. This simple example demonstrates how most workloads involving many-to-many relationships are hard to partition. Such relationships occur in many places, e.g., order processing applications where orders are issued to suppliers or brokers that service many customers (as modeled by TPC-E) or message boards where users post on multiple forums (with queries to search for posts by forum or user).

One solution to this problem is to use a *fine-grained partitioning* strategy, where tuples are allocated to partitions in a way that exploits relationships between records. In our social networking example, a user and his or her friends can be co-located such that some queries go to just one partition. Thus, a careful assignment of tuples to partitions can reduce or eliminate distributed transactions, allowing a workload to be efficiently scaled across multiple machines.

A second problem with traditional partitioning is that while queries on the partitioning attribute go to a single partition, queries on other attributes must be broadcast to all partitions. For example, for Wikipedia, two of the most common queries select an article either by a numeric id or by title. No matter which attribute is chosen for partitioning, the queries on the other attribute need to be sent to all partitions. What is needed to address this is a *partition index* that specifies which partitions contain tuples matching a given attribute value (e.g., article id), without partitioning the data by those attributes. Others have proposed using distributed secondary indexes to solve this problem, but they require two network round trips to two different partitions [2]. Another alternative are multi-attribute indexes, which require accessing a fraction of the partitions instead of all of them (e.g. $\sqrt{N}$ for two attributes) [3]. While this is an improvement, it still requires more work as the system grows, which hampers scalability.

To solve both the fine-grained partitioning and partition index problems, we introduce *lookup tables*. Lookup tables map from a *key* to a set of partition ids that store the corresponding tuples. This allows the administrator to partition tuples in an arbitrary (fine-grained) way. Furthermore, lookup tables can be used as partition indexes, since they specify where to find a tuple with a given key value, even if the table is not partitioned according to that key. A key property is that with modern systems, lookup tables are small enough that they can be cached in memory on database query routers, even for very large databases. Thus, they add minimal overhead when routing queries in a distributed database.

To evaluate this idea, we built a database front-end that uses lookup tables to execute queries in a shared-nothing distributed

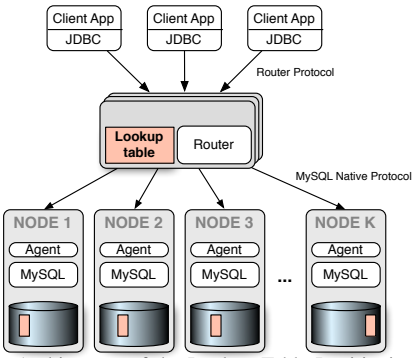

Fig. 1. Architecture of the Lookup Table Partitioning System

database. In our design, *query routers* send queries to backend databases (unmodified MySQL instances, in our case), which host the partitions. Lookup tables are stored in memory, and consulted to determine which backends should run each query. We show that this architecture allows us to achieve linear scale-out on several difficult to partition datasets: a TPC-E-like transactional benchmark, a Wikipedia benchmark using real data from January 2008 (≈3M tuples), and a snapshot of the Twitter social graph from August 2009 (≈4B tuples). Compared to hash or range-partitioning, we find that lookup tables can provide up to a factor of 3 greater throughput on a 10 node cluster. Furthermore, both hash and range partitioning achieve very limited scale-out on these workloads, suggesting that fine-grained partitioning is necessary for them to scale.

Though lookup tables are a simple idea, making them work well involves a number of challenges. First, lookup tables must be stored compactly in RAM, to avoid adding additional disk accesses when processing queries. In this paper, we compare several representations and compression techniques that still allow efficient random access to the table. On our datasets, these techniques result in 3× to 250× compression without impacting query throughput. A second challenge is efficiently maintaining lookup tables in the presence of updates; we describe a simple set of techniques that guarantee that query routers and backend nodes remain consistent, even in the presence of failures.

Finally, we note that this paper is not about *finding* the best fine-grained partitioning. Instead, our goal is to show that lookup tables can improve the performance of many different workloads. We demonstrate that several existing partitioning schemes [1], [4], as well as manual partitioning can be implemented with lookup tables. They result in excellent performance, providing 40% to 300% better throughput on our workloads than either range or hash partitioning and providing better scale-out performance.

Next we provide a high level overview of our approach, including the system architecture and a discussion of the core concept behind lookup tables.

## II. Overview

The structure of our system is shown in Fig. 1. Applications interact with the database as if it is a traditional single system, using our JDBC driver. Our system consists of two layers: *backend* databases, which store the data, and *query routers* that contain the lookup table and partitioning metadata. Each backend runs an unmodified DBMS (MySQL in our prototype), plus an *agent* that manages updates to the lookup tables, described in Section IV. Routers receive an application query and execute it by distributing queries to the appropriate backends. We use a typical presumed abort two-phase commit protocol with the read-only participant optimization to manage distributed transactions across backends, using the XA transactions supported by most SQL databases. This design allows the throughput to be scaled by adding more machines. Adding backends requires the data to be partitioned appropriately, but adding routers can be done on demand; they do not maintain any durable state, except for distributed transaction commit logs. Routers process queries from multiple clients in parallel, and can also dispatch sub-queries to backends in parallel. Thus, a single router can service many clients and backends, but more can be added to service additional requests.

The routers are given the network address for each backend, the schema, and the partitioning metadata when they are started. The partitioning metadata describes how data is divided across the backends, which can either be traditional hash or range partitioning, or lookup tables. To simplify our initial description, we describe our basic approach assuming that the lookup table does not change, and that each router has a copy of it when it starts. We discuss how routers maintain their copy of the lookup table when data is inserted, updated, and deleted in Section IV.

To describe the basic operation of lookup tables, consider the social networking example from the introduction. This example contains two tables: `users` and `followers`, each identified by an integer primary key. The `followers` relation contains two foreign keys to the `users` table: `source` and `destination`. This creates a many-to-many relationship, from each user to the users that they follow. The `users` table contains a status field. Users want to get the status for all users they are following. This is done with the following two SQL queries, the first of which fetches the list of user ids that are followed and the second of which fetches the status.

```
R=SELECT destination FROM followers WHERE source=x
SELECT * FROM users WHERE id IN (R)
```

For traditional hash partitioning, we can partition the `users` table by id, and partition the `followers` table by source. For lookup tables, we can partition `users` such that related users (e.g., that share many friends) are stored on the same node, and use the same lookup table to partition `followers` by source. We evaluate how effective this type of clustering can be with real data in Section VI-A; in that case, we use various partitioning schemes to figure out how to initially place `users`. As noted above, our goal in this paper is not to propose new partitioning strategies, but to show that lookup tables are flexible enough to implement any of a number of strategies and can lead to much better performance than simple hash or range partitioning.

### A. Basic Lookup Table Operation

When a router receives a query from the application, it must determine which backends store the data that is referenced. For queries referencing a column that uses a lookup table (e.g., `SELECT destination FROM followers`

WHERE source = 42), the router consults its local copy of the lookup table and determines where to send the query. In the case of simple equality predicates, the query can simply be passed through to a single backend. If multiple backends are referenced (e.g., via an IN clause) then the query is rewritten and a separate query is sent to each backend. The results must then be merged in an appropriate way (e.g., via unions, sorts, or aggregates) before being returned to the client. More complex queries may require multiple rounds of sub-queries. Our query routing protocol is discussed in Section III-B.

For our example, the first query is sent to a single partition based on the lookup table value for user id $= x$. The second query is then sent to a list of partitions, based on the user ids returned by the first query. For each user id, the destination partition is found in the table. This gives us the freedom to place users on any partition. In this case, ideally we can determine an intelligent partitioning that co-locates a user with the users they are following. As the data is divided across more machines, the number of partitions that is accessed by this second query should stay small.

For hash partitioning, the second query accesses several partitions, since the users are uniformly distributed across partitions. When we attempt to scale this system by adding more machines, the query accesses more partitions. This limits the scalability of this system. We measure the coordination overhead of such distributed queries in Section VI.

### B. Storing Lookup Tables

In order to use lookup tables to route queries, they must be stored in RAM at each router, in order to avoid imposing any performance penalty. Conceptually, a lookup table is a map between a value and a list of partitions where matching values are stored. The common case is mapping a unique primary integer key to a partition. The straightforward representation for this case consists of an in-memory map from an 8-byte integer key to a 2 or 4-byte integer partition id for each tuple, requiring at least 10 bytes per tuple to be stored in memory, plus additional overhead for the data structure. Unfortunately, this implementation is impractical for large tables with trillions of tuples, unless we want to require our frontend nodes to have terabytes of RAM. Hence, in Section V we present a number of implementation techniques that allow large lookup tables to be stored efficiently in RAM.

## III. Lookup Table Query Processing

In this section, we describe how lookup tables are defined and used for distributed query planning.

### A. Defining Lookup Tables

We begin with some basic SQL syntax a database administrator can use to define and populate lookup tables. These commands define the metadata used by the query router to perform query execution. To illustrate these commands, we again use our example users and followers tables.

First, to specify that the users table should be partitioned into part1 and part2, we can write:

```
CREATE TABLE users (
  id int, ..., PRIMARY KEY (id),
  PARTITION BY lookup(id) ON (part1, part2)
  DEFAULT NEW ON hash(id));
```

This says that users is partitioned with a lookup table on id, and that new tuples should be added by hashing on id. Here, we require that the partitioning attribute be unique (i.e., each tuple resides on only one backend.) The metadata about the assignment of logical partitions to physical node is maintained separately to allow physical independence.

We can also explicitly place one or more users into a given partition, using ALTER TABLE:

```
ALTER TABLE users SET PARTITION=part2 WHERE id=27;
```

We also allow tuples to be replicated, such that a given id maps to more than one partition, by specifying a list of partitions in the SET PARTITION = clause.

Here, the WHERE clause can contain an arbitrary expression specifying a subset of tuples. Additionally, we provide a way to load a lookup table from an input file that specifies a mapping from the partitioning key to the logical partition (and optionally the physical node) on which it should be stored (this makes it easy to use third party partitioning tools to generate mappings and load them into the database.)

To specify that the followers table should be partitioned in the same way as the users table, we can create a *location dependency* between the two tables, as follows (similar syntax is used in the table definition for the followers table):

```
ALTER TABLE followers
  PARTITION BY lookup(source) SAME AS users;
```

This specifies that each followers tuple f should be placed on the same partition as the users tuple u where u.id = f.source. This is important for joins between the followers and users table, since it guarantees that all matching tuples will reside in the same partition—this enables more effective join strategies. Also, this enables *reuse*, meaning that we only need to keep one copy of the lookup table for both attributes.

Finally, it is possible to define partition indexes (where a lookup table is defined on table that is already partitioned in some other way) using the CREATE SECONDARY LOOKUP command, as follows:

```
CREATE SECONDARY LOOKUP l_a ON users(name);
```

This specifies that a lookup table l_a should be maintained. This allows the router(s) to figure out which logical partition (and thus physical node) a given user resides on when presented with a query that specifies that user's name. Secondary lookups may be non-unique (e.g., two user's may be named 'Joe' and be on different physical backends.)

### B. Query Planning

Distributed query planning is a well-studied problem, therefore in the following we limit our discussion to clarify some non-obvious aspects of query planning over look-up tables.

In order to plan queries across our distributed database, each router maintains a copy of the partitioning metadata defined by the above commands. This metadata describes how each table is partitioned or replicated, which may include dependencies between tables. This metadata does not include the lookup table itself, which is stored across all partitions. In this work, we assume that the database schema and partitioning strategy

do not change. However, the lookup table itself may change, for example by inserting or deleting tuples, or moving tuples from one partition to another.

The router parses each query to extract the tables and attributes that are being accessed. This list is compared with the partitioning strategy. The goal is to push the execution of queries to the backend nodes, involving as few of them as possible. As a default, the system will fetch all data required by the query (i.e., for each table, run a select query to fetch the data from each backend node), and then execute the query locally at the router. This is inefficient but correct. In the following, we discuss simple heuristics that improve this, by pushing predicates and query computation to the backend nodes for many common scenarios. The heuristics we present cover all of the queries from our experimental datasets.

*Single-table queries:* For single table queries, the router first identifies the predicates that are on the table's partitioning key. The router then consults the lookup table to find which backends tuples that match this predicate reside on, and generates sub-queries from the original query such that each can be answered by a single backend. The sub-queries are then sent and results are combined by the appropriate operation (union, sort, etc) before returning the results to the user. For equality predicates (or `IN` expressions), it is straightforward to perform this lookup and identify the matching backends. For range queries, broadcasting the query to all backends is the default strategy. For countable ranges, such as finite integer ranges, it is possible to reduce the number of participants by looking up each value in the range and adding each resulting partition to the set of participants. For uncountable ranges, such as variable length string ranges, it is impossible to look up each value, so the router falls back to a broadcast query.

Updates typically touch a single table, and are therefore treated similarly to single-table queries. The exception is that when updating a partitioning key, the tuple may actually need to be moved or a lookup table updated. The router detects these cases and handles them explicitly. Inserts, moves, and deletes for lookup tables are described in Section IV.

*Multi-table queries:* Joins can only be pushed down if the two tables are partitioned in the same way (e.g., on the same attribute, using the same lookup table.) In this case, one join query is generated for each backend, and the result is UNION'ed at the router. In many OLTP workloads, there will be additional single-table equality predicates that cause results to be only produced at one backend (e.g., if we are looking up the message of a particular user in our social networking application, with users and messages both partitioned using the same lookup table on `users.id`). We detect this case and send the query only to one backend.

For joins over tables partitioned on different attributes (which do not arise in our test cases) we evaluate these by collecting tuples from each backend that satisfy the non-join predicates, and evaluating the join at the router. This is clearly inefficient, and could be optimized using any of the traditional strategies for evaluating distributed joins [5]. These strategies, as well as more complex queries, such as nested expressions, may result in multiple rounds of communication between the router and backends to process a single query.

## IV. Start-up, Updates and Recovery

In the previous section, we assumed that routers have a copy of the lookup table when they start, and that the table does not change. In this section, we relax these assumptions and describe how our system handles start-up, updates, and recovery. When starting, each router must know the network address of each backend. In our research prototype this is part of the static configuration data for each router, but a production implementation would instead load this from a metadata service such as Zookeeper [6], allowing partitions to be added, moved, and removed. The router then attempts to contact other routers to copy their lookup table. As a last resort, it contacts each backend agent to obtain the latest copy of each lookup table subset. The backend agent must scan the appropriate tables to generate the set of keys stored on that partition. Thus, there is no additional durable state. This does not affect the recovery time, as this scanning is a low-priority background task that is only needed when new routers start.

After the initial lookup table is loaded on the routers, it may become stale as data is inserted, deleted and moved between backends. To ensure correctness, the copy of the lookup table at each router is considered a cache that may not be up to date. This means that routers only store soft state, allowing them to be added or removed without distributed coordination. To keep the routers up to date, backends piggyback changes with query responses. However, this is only a performance optimization, and is not required for correctness.

Lookup tables are usually *unique*, meaning that each key maps to a single partition. This happens when the lookup table is on a unique key, as there is only one tuple for a given key, and thus only one partition. However, it also happens for non-unique keys if the table is partitioned on the lookup table key. This means there can be multiple tuples matching a given value, but they are all stored in the same partition. This is in contrast to *non-unique* lookup tables, where for a given value there may be multiple tuples located on multiple partitions.

For unique lookup tables, the existence of a tuple on a backend indicates that the query was routed correctly because there cannot be any other partition that contains matching tuples. If no tuples are found, we may have an incorrect lookup table entry, and so fall back to a broadcast query. This validation step is performed by the router. The pseudocode is shown in Fig. 2. There are three cases to consider: the router's lookup table is up to date, the table is stale, or there is no entry.

**Up to date:** The query goes to the correct destination and at least one tuple is found, so the router knows the lookup table entry is correct.

**Stale lookup table entry:** The query is sent to a partition that used to store the tuple, but the tuple has been deleted or moved. In either case, the tuple is not found so the router will fall back to broadcasting the query everywhere. This is guaranteed to find moved tuples, or find no tuple if it has been deleted.

**No lookup table entry:** The query is first sent to the default partition based on the key (as defined by the `DEFAULT NEW` expression in the table definition—see

Section III-A). If the tuple is found, then the correct answer is returned. If a tuple is not found, a broadcast query is required to locate the tuple and update the lookup table.

This simple scheme relies on expensive broadcast queries when a tuple is not found. This happens for the following classes of queries:

- *Queries for keys that do not exist.* These queries are rare for most applications, although they can happen if users mis-type, or if external references to deleted data still exist (e.g., web links to deleted pages).
- *Inserts with an explicit primary key on tables partitioned using a lookup table on the primary key.* These inserts need to query all partitions to enforce uniqueness, since a tuple with the given key could exist on any partition. However, auto-increment inserts can be handled efficiently by sending them to any partition, and allowing the partition to assign an unused id. This is the common case for the OLTP and web workloads that we target. Foreign key constraints are typically not enforced in these workloads due to the performance cost. However, lookup tables do not change how they are implemented by querying the appropriate indexes of other tables. Thus, they can be supposed if desired.
- *Queries for recently inserted keys.* We reduce the probability this occurs by pushing lookup table updates to routers on a "best effort" basis, and by piggybacking updates along with other queries.

This simple scheme is efficient for most applications, since the vast majority of queries can be sent to a single partition. If desired, we can more efficiently handle queries for missing and deleted tuples by creating "tombstone" records in the backend databases. Specifically, we can create records for the next $N$ unassigned ids, but mark them as deleted via a special "deleted" column in the schema.

When a statement arrives for one of these ids it will be sent to exactly one partition. In case of inserts, deleted tuples will be marked as "live," and the existing values replaced with the new values. In case of other queries, the value of the "deleted" column will be checked to see if the application should see the data. Similarly, when deleting tuples, the tuple is marked as deleted, without actually removing it. This ensures that queries for this deleted tuple continue to be directed to the correct partition. Eventually, very old tuples that are no longer queried can be actually removed. This "tombstone" approach is a simple way to handle these problematic queries.

### A. Non-Unique Lookup Tables

Non-unique lookup tables are uncommon, but arise for two reasons. First, a lookup table on a non-unique attribute that is not used to partition the table will map a single value to multiple partitions. Second, an application may choose to replicate certain tuples across multiple partitions to make reads more efficient, while making updates more expensive. The previous protocol relies on the fact that if at least one tuple is found, that backend server must contain all tuples for the given key, and thus the correct answer was returned. However, with non-unique lookup tables, tuples can be found even if some partitions are incorrectly omitted. It is always correct to query all partitions, but that is also very expensive.

To verify that all tuples matching a given value were found without querying all partitions, we designate one partition for each value as the primary partition. This partition records the set of partitions that store tuples matching the value. This information is maintained in a transactionally consistent fashion by using distributed transactions when new partitions are added to the set or existing partitions are removed. This data is persistently stored in each backend, as an additional column in the table. Since there is a unique primary partition for each value, we use a protocol similar to the previous one to ensure that the router finds it. When the primary partition is found, the router can verify that the correct partitions were queried. Secondary partitions store the identity of the current primary. On any query, the primary returns the current list of partitions for the given value, so the router can easily verify that it queried the correct set of partitions. If not, it can send the query to the partitions that were missed the first time.

The common case for this protocol is that the lookup table is up to date, so every statement is directed to the correct set of partitions and no additional messages are required. For inserts or deletes, the typical case is that the partition contains other tuples with the lookup table value, and thus the primary partition does not need to be updated. Thus, it is only in the rare cases where the partition set changes that the primary partition needs updating.

The details of query execution on non-unique lookup tables is shown in the routerStatement procedure in Fig. IV-A (inserts and deletes are described below). If an entry exists, the router first sends the query to the set of partitions cached in its lookup table. After retrieving the results, it looks for a partition list from the primary. If a list is returned, the router calculates the set of partitions that were missing from its first round of queries. If the lookup table was up to date, then the set of missing partitions will be empty. If there is no primary response in the initial round of responses, then the lookup table was incorrect and the set of missing partitions is set to all remaining partitions. Finally, the router queries the set of missing partitions and combines the results from all partitions. This ensures that all partitions with a given value are queried, even if the router's information is incorrect.

Inserts and deletes must keep the primary partition's list up to date. The router does this as part of the transaction that includes the insert or delete. The backend agent detects when the first tuple for a value is added to a partition, or when the only tuple for a value is deleted. It return an indication that the primary must be updated to the router. When the router receives this message, it adds or deletes the backend from the partition list at the primary. If the last tuple is being removed from the primary, then the router selects a secondary partition at random to become the new primary and informs all secondary partitions of the change. Since this is performed as part of a distributed transaction, failure handling and concurrent updates are handled correctly. The insert protocol is shown in the routerInsert and backendInsert procedures

```
function executeStatement(statement):
  // find partition in lookup table, or default partition
  lookupKey = parse lookup table key from statement
  if lookup table entry for lookupKey exists:
    destPart = lookup table entry for lookupKey
  else if lookup table has a defaultPartitionFunction:
    destPart = defaultPartitionFunction(lookupKey)
  if destPart is not null:
    resultSet = execute statement on destPart
    if statement matched at least one tuple:
      // correct partition in the lookup table
      return resultSet

  // wrong partition or no entry in the lookup table
  resultSets = broadcast statement to all partitions
  for partitionId, resultSet in resultSets:
    if statement matched at least one tuple:
      // found the correct partition
      set lookup table key lookupKey → partitionId
      return resultSet

  // no tuple: return an empty result set
  remove lookupKey from lookup table, if it exists
  return first resultSet from resultSets
```

Fig. 2. Lookup table validation for unique lookup tables

```
function nonUniqueRouterStatement(statement):
  lookupKey = get lookup table key from statement
  missingParts = all partitions
  results = ∅
  if lookup table entry for lookupKey exists:
    partList = entry for lookupKey
    results = send statement to partList
    primaryFound = false
    for result in results:
      if result.partList is not empty:
        primaryFound = true
        missingParts = result.partList − partList
        break
    if not primaryFound:
      // the lookup table is incorrect
      missingParts = all partitions - partList:
      remove lookupKey from the lookup table
  results = results ∪ send statement to missingParts
  for result in results:
    if result.partList is not empty:
      // not strictly needed if we were already up-to-date
      set lookup table lookupKey → result.partList
      break
  return results

function nonUniqueRouterInsert(tuple, backend):
  // use partitioning key for insert and check constraints
  result, status = insert tuple as usual
  if status is no primary update needed:
    return result
  lookupKey = extract lookup table key from tuple
  primary = lookup table entry for lookupKey
  result = send (lookupKey, backend) to primary
  if result is success:
    return result
  // failed update or no entry: broadcast
  results = broadcast (lookupKey, backend) mapping
  if results does not contain a success result:
    make backend the primary for lookupKey
  return result

function nonUniqueBackendInsert(tuple):
  status = no primary update needed
  result = insert tuple
  if result is success:
    lookupKey = lookup key from tuple
    keyCount = # tuples where value == lookupKey
    if keyCount == 1:
      status = primary update needed
  return result, status
```

Fig. 3. Lookup table validation for non-unique lookup tables

in Fig. IV-A. The delete protocol (not shown) works similarly.

## V. Storage Alternatives

A lookup table is a mapping from each distinct value of a field of a database table to a logical partition identifier (or a set if we allow lookups on non-unique fields). In our current prototype we have two basic implementations of lookup tables: hash tables and arrays. Hash tables can support any data type and sparse key spaces, and hence are a good default choice. Arrays work better for dense key-spaces, since hash tables have some memory overhead. For arrays, we use the attribute value as the array offset, possibly modified by an offset to account for ids that do not start at zero. This avoids explicitly storing the key and has minimal data structure overhead, but becomes more wasteful as the data grows sparser, since some keys will have no values.

We ran a series of experiments to test the scalability of these two implementations. We found that the throughput of arrays is about 4x the throughput of hash tables, but both implementations can provide greater than 15 million lookups per second, which should allow a single front-end to perform routing for almost any query workload. Arrays provide better memory utilization for dense key-spaces, using about 10x less RAM when storing dense, 16-bit keys. However, arrays are not always an option because they require mostly-dense, countable key spaces (e.g., they cannot be used for variable length strings). To test the impact of key space density, we compared our implementations on different key-space densities. When density falls below 40-50% the hash-map implementation becomes more memory efficient than an array-based one.

### A. Lookup Table Reuse

Whenever location dependencies over fields that are partitioned by lookup tables, it means that two (or more) tables are partitioned using an identical partitioning strategy. Therefore, a simple way to reduce the memory footprint of lookup tables is to reuse the same lookup table in the router for both tables. This reduces main memory consumption and speeds up the recovery process, at the cost of a slightly more complex handling of metadata.

### B. Compressed Tables

We can compress the lookup tables in order to trade CPU time to reduce space. Specifically, we used Huffman encoding, which takes advantage of the skew in the frequency of symbols (partition identifiers). For lookup tables, this skew comes from two sources: (i) partition size skew, (e.g. due to load balancing some partitions contain fewer tuples than others), and (ii) range affinity (e.g., because tuples inserted together tend to be in the same partition). This last form of skew can be leveraged by "bucketing" the table and performing separate Huffman encoding for each bucket.

This concept of bucketing is similar to the adaptive encodings used by compression algorithms such as Lempel-Ziv. However, these adaptive encodings require that the data be decompressed sequentially. By using Huffman encoding directly, we can support random lookups by maintaining a sparse index on each bucket. The index maps a sparse set of keys to their corresponding offsets in the bucket. To perform a lookup of a tuple id, we start from the closest indexed key smaller than the desired tuple and scan forward.

We tested this bucketed Huffman compression on Wikipedia, Twitter, and TPC-E data (details of these data sets are in Section VI-A). The compression heavily depends on the skew of the data, for Wikipedia and Twitter we only have modest skew, and we obtain (depending on the bucketing) compression between 2.2× and 4.2× for Wikipedia and 2.7× to 3.7× for Twitter. For TPC-E there is a very strong affinity between ranges of tuples and partition ids, and therefore we obtain a dramatic 250× compression factor (if TPC-E didn't have this somewhat artificial affinity, bucketing performance would be closer to Wikipedia and Twitter.) We found that although Huffman encoding slightly increases router CPU utilization, a single router was still more than sufficient to saturate 10 backends. Furthermore, our architecture easily supports multiple routers, suggesting that the CPU overhead of compression is likely worthwhile.

### C. Hybrid Partitioning

Another way of reducing the memory footprint of a lookup table is to combine the fine-grained partitioning of a lookup table with the space-efficient representation of range or hash

partitioning. In effect, this treats the lookup table as an exception list for a simpler strategy. The idea is to place "important" tuples in specific partitions, while treating the remaining tuples with a default policy.

To derive a hybrid partitioning, we use decision tree classifiers to generate a rough range partitioning of the data. To train the classifier, we supply a sample of tuples, labeled with their partitions. The classifier then produces a set of intervals that best divide the supplied tuples (according to their attribute values) into the partitions. Unlike how one would normally use a classifier, we tune the parameters of the learning algorithm cause over-fitting (e.g., we turn off cross-validation and pruning). This is because in this context we do not want a good generalization of the data, but rather we want the decision tree classifier to create a more compact representation of the data. The trained decision tree will produce correct partitions for a large fraction of the data, while misclassifying some tuples. We use the set of predicates produced by the decision tree as our basic range partitioning (potentially with many small ranges), and build a lookup table for all misclassifications.

The net effect is that this hybrid partitioning correctly places all tuples in the desired partitions with a significant memory savings. For example, on the Twitter dataset the decision tree correctly places about 76% of the data, which produces almost a 4× reduction in memory required to store the lookup table.

One advantage of this application of decision trees is that the runtime of the decision tree training process on large amounts of data is unimportant for this application. We can arbitrarily subdivide the data and build independent classifiers for each subset. This adds a minor space overhead, but avoids concerns about the decision tree classifier scalability.

### D. Partial Lookup Tables

So far we have discussed ways to reduce the memory footprint while still accurately representing the desired fine-grained partitioning. If these techniques are not sufficient to handle a very large database, we can trade memory for performance, by maintaining only the recently used part of a lookup table. This can be effective if the data is accessed with skew, so caching can be effective. The basic approach is to allow each router to maintain its own least-recently used lookup table over part of the data. If the id being accessed is not found in the table, the router falls back to a broadcast query, as described in Section IV, and adds the mapping to its current table. This works since routers assume their table may be stale, thus missing entries are handled correctly.

We use Wikipedia as an example to explain how partial lookup tables can be used in practice. Based on the analysis of over 20 billion Wikipedia page accesses (a 10% sample of 4 months of traffic), we know that historical versions of an article, which represent 95% of the data size, are accessed a mere 0.06% of the time [7]. The current versions are accessed with a Zipfian distribution. This means that we can properly route nearly every query while storing only a small fraction of the ids in a lookup table. We can route over 99% of the queries for English Wikipedia using less than 10-15MB of RAM for lookup tables, as described in Section VI. A similar technique can be applied to Twitter, since new tweets are accessed much more frequently that historical tweets.

Although we tested the above compression techniques on all our datasets, in the results in Section VI, we use uncompressed lookup tables since they easily fit in memory.

TABLE I
EXPERIMENTAL SYSTEMS

| Num. Machines | Description |
|---|---|
| 10 Backends | 1 × Xeon 3.2 GHz, 2 GB RAM, 1 × 7200 RPM SATA |
| 2 Client/Router | 2 × Quad-Core Xeon E5520 2.26GHz, 24 GB RAM, 6 × 7200 RPM SAS (5 ms lat.), HW RAID 5 |
| All | Linux Ubuntu Server 10.10, Sun Java 1.6.0_22-b04, MySQL 5.5.7 |

TABLE II
WORKLOAD SUMMARY

| **Data Set / Transactions** | **Fraction** | **Distribution Properties** | | | | |
|---|---|---|---|---|---|---|
| | | Hashing | | Range | | Lookup |
| | | *Broadcast* | *2PC* | *Broadcast* | *2PC* | *Broadcast 2PC* |
| **Wikipedia** | | | | | | |
| Fetch Page | 100% | ✗ | ✗ | ✗ | ✗ | |
| **Twitter** | | | | | | |
| Insert Tweet | 10% | | | | | ✗ |
| Tweet By Id | 35% | | | | | |
| Tweets By User | 10% | ✗ | ✗ | ✗ | ✗ | |
| Tweets From Follows | 40% | ✗ | ✗ | ✗ | ✗ | |
| Names of Followers | 5% | | ✗ | | ✗ | ✗ |
| **TPC-E** | | | | | | |
| Trade Order | 19.4% | | ✗ | | | |
| Trade Result | 19.2% | ✗ | ✗ | ✗ | ✗ | |
| Trade Status | 36.5% | | ✗ | | | |
| Customer Position | 25.0% | ✗ | ✗ | ✗ | ✗ | |

## VI. EXPERIMENTAL EVALUATION

In order to evaluate the benefit of using lookup tables, we ran a number of experiments using several real-world datasets (described below). We distributed the data across a number of backend nodes running Linux and MySQL. The detailed specifications for these systems are listed in Table I. The backend servers we used are older single-CPU, single-disk systems, but since we are interested in the relative performance differences between our configurations, this should not affect the results presented here. Our prototype query router is written in Java, and communicates with the backends using MySQL's protocol via JDBC. All machines were connected to the same gigabit Ethernet switch. We verified that the network was not a bottleneck in any experiment. We use a closed loop load generator, where we create a large number of clients that each send one request at a time. This ensures there is sufficient concurrency to keep each backend fully utilized.

### A. Datasets and Usage Examples

In this section, we the real-world data sets we experiment with, as well as the techniques we used to partition them using both lookup tables and hash/range partitioning. To perform partitioning with lookup tables, we use a combination of manual partitioning, existing partitioning tools and semi-automatic fine-grained partitioning techniques developed for these data sets. This helps us demonstrate the flexibility of lookup tables for supporting a variety of partitioning schemes/techniques.

The schemas and the lookup table partitioning for all the workloads are shown in Fig. 4. In this figure, a red underline indicates that the table is partitioned on the attribute. A green highlight indicates that there is a lookup table on the attribute. A solid black arrow indicates that two attributes have a *location dependency* and are partitioned using the `SAME AS` clause and share a lookup table. A dashed arrow indicates that

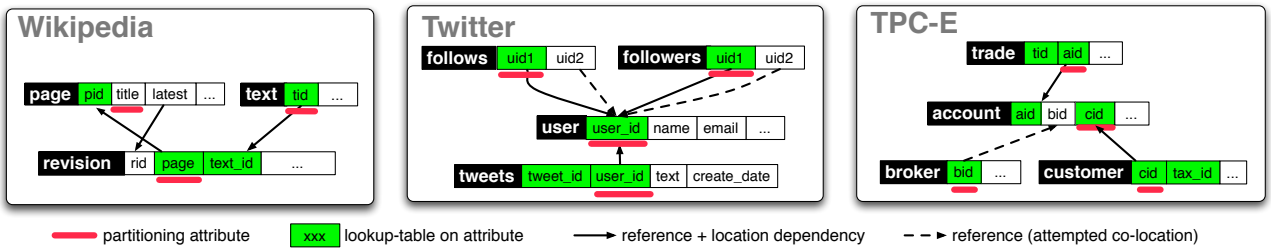


Fig. 4. Schemas and lookup table partitioning for: Wikipedia, Twitter, TPC-E

the partitioning scheme attempts to co-locate tuples on these attributes, but there is not a perfect location dependency.

The transactions in each workload are summarized in Table II. A mark in the "Broadcast" column indicates that the transaction contains queries that must be broadcast to all partitions for that partitioning scheme. A mark in the "2PC" column indicates a distributed transaction that accesses more than one partition and must use two-phase commit. Both these properties limit scalability. The details of these partitioning approaches are discussed in the rest of this section.

In summary, Wikipedia and TPC-E show that lookup tables are effective for web-like applications where a table is accessed via different attributes. Twitter shows that it can work for difficult to partition many-to-many relationships by clustering related records. Both TPC-E and Wikipedia contain one-to-many relationships.

**Wikipedia:** We extracted a subset of data and operations from Wikipedia. We used a snapshot of English Wikipedia from January 2008 and extracted a 100k page subset. This includes approximately 1.5 million entries in each of the revision and text tables, and occupies 36 GB of space in MySQL. The workload is based on an actual trace of user requests from Wikipedia [7], from which we extracted the most common operation: fetch the current version of an article. This request involves three queries: select a page id (`pid`) by `title`, select the page and revision tuples by joining the page and revision on `revision.page = page.pid` and `revision.rid = page.latest`, then finally select the text matching the text id from the revision tuple. This operation is implemented as three separate queries even though it could be one because of changes in the software over time, MySQL-specific performance issues, and the various layers of caching outside the database that we do not model.

The partitioning we present was generated by manually analyzing the schema and query workload. This is the kind of analysis that developers do today to optimize distributed databases. We attempted to find a partitioning that reduces the number of partitions involved in each transaction and query. We first consider strategies based on hash or range partitioning.

**Alternative 1:** Partition `page` on `title`, `revision` on `rid`, and `text` on `tid`. The first query will be efficient and go to a single partition. However, the join must be executed in two steps across all partitions (fetch `page` by `pid` which queries all partitions, then fetch `revision` where `rid` = `p.latest`). Finally, `text` can be fetched directly from one partition, and the read-only distributed transaction can be committed with another broadcast to all partitions (because of the 2PC read-only optimization). This results in a total of $2k + 3$ messages.

**Alternative 2:** Partition `page` on `pid`, `revision` on `page` = `page.pid`, and `text` on `tid`. In this case the first query goes everywhere, the join is pushed down to a single partition and the final query goes to a single partition. This results in a total of $2k + 2$ messages.

The challenge with this workload is that the `page` table is accessed both by `title` and by `pid`. Multi-attribute partitioning, such as MAGIC [3], is an alternative designed to help with this kind of workload. Using MAGIC with this workload would mean that the first query would access $\sqrt{k}$ partitions of the table, while the second query would access a partially disjoint $\sqrt{k}$ partitions. The final query could go to a single partition. This still requires a distributed transaction, so there would be a total of $4\sqrt{k} + 1$ messages. While this is better than both hash and range partitioning, this cost still grows as the size of the system grows.

Lookup tables can handle the multi-attribute lookups without distributed transactions. We hash or range partition `page` on `title`, which makes the first query run on a single partition. We then build a lookup table on `page.pid`. We co-locate revisions together with their corresponding page by partitioning `revision` using the lookup table (`revision.page` = `page.pid`). This ensures that the join query runs on the same partition as the first query. Finally, we create a lookup table on `revision.text_id` and partitioning on `text.tid` = `revision.text_id`, again ensuring that all tuples are located on the same partition. This makes every transaction execute on a single partition, for a total of 4 messages. With lookup tables, the number of messages does not depend on the number of partitions, meaning this approach can scale when adding machines. The strategy is shown in Fig. 4, with the primary partitioning attribute of a table underlined in red, and the lookup table attributes highlighted in green.

We plot the expected number of messages for these four schemes in Fig. 5 The differences between these partitioning schemes become obvious as the number of backends increases. The number of messages required for hash and range partitioning grows linearly with the number of backends, implying that this solution will not scale. Multi-attribute partitioning (MAGIC) scales less than linearly, which means that there will be some improvement when adding more backends. However, lookup tables enable a constant number of messages for growing number of backends and thus better scalability.

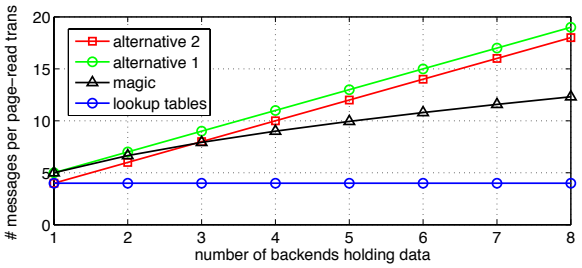


Fig. 5. Wikipedia, comparison of alternative solutions.

Exploiting the fact that lookup tables are mostly dense integers (76 to 92% dense), we use an array implementation of lookup tables, described in detail in Section V. Moreover, we reuse lookup tables when there are location dependencies. In this case, there is one a lookup table shared for both `page.pid` and `revision.page`, and a second table for `revision.text_id` and `text.tid`. Therefore, we can store the 360 million tuples in the complete Wikipedia snapshot in less than 200MB of memory, which easily fits in RAM. This dataset allows us to verify lookup table scalability to large databases, and demonstrate enable greater scale out.

**Twitter:** Lookup tables are useful for partitioning complex social network datasets, by grouping "clusters" of users together. In order to verify this, we obtained a complete and anonymized snapshot of the Twitter social graph as of August 2009 containing 51 million users and almost 2 billion follow relationships [8]. We replicate the follow relationship to support lookups in both directions by an indexed key in order to access more efficiently who is being followed as well as users who are following a given user. This is the way Twitter organizes their data, as of 2010 [9]. The schema and the lookup table based partitioning is shown in Fig. 4.

To simulate the application, we synthetically generated tweets and a query workload based on properties of the actual web site [9]. Our read/write workload consists of the following operations: 1) insert a new tweet, 2) read a tweet by `tweet_id`, 3) read the 20 most recent tweets of a certain user, 4) get the 100 most recent tweets of the people a user follows, and 5) get the names of the people that follow a user. Operations 4 and 5 are implemented via two separate queries. The result is a reasonable approximation of a few core features of Twitter. We use this to compare the performance of lookup tables to hash partitioning, and to show that we can use lookup tables with billions of tuples.

We partitioned the Twitter data using hash partitioning and lookup tables. We do not consider multi-attribute partitioning here because most of these accesses are by `tweet_id`, and those would become requests to $\sqrt{k}$ partitions, greatly reducing performance. For hash partitioning, we simply partitioned the tweets table on `id` and the rest of the tables on `user_id`, which we discuss in Section VI-D.

For lookup tables, we also partition on `user_id`, but we carefully co-locate users that are socially connected. This dataset allows us to showcase the flexibility of lookup tables presenting three different partitioning schemes:

1) A lookup-table based heuristic partitioning algorithm that clusters users together with their friends, and only replicates extremely popular users— we devised a very fast and greedy partitioner that explores each edge in the social graph only once and tries to group user, while balancing partitions. This represents an ad hoc user provided heuristic approach.
2) Replicating users along with their friends, as proposed in work on *one hop replication* by Pujol at el. [4]. This shows how lookup tables can be used to implement state-of-the-art partitioning strategies.
3) A load-balanced version of 2 that attempts to make each partition execute the same number of queries, using detailed information about the workload. This shows how lookup tables can support even manually-tuned partitioning strategies. This is identical to what the Schism automatic partitioner, our previous work, would do [1].

Schemes 2 and 3 ensure that all queries are local while inserts can be distributed due to replication. The lookup table contains multiple destination partitions for each user. Scheme 3 extends scheme 2 by applying a greedy load balancing algorithm to attempt to further balance the partitions.

**TPC-E:** TPC-E is a synthetic benchmark designed to simulate the OLTP workload of a stock brokerage firm [10]. It is a relatively complex benchmark, composed of 12 transaction types and 33 tables. For the sake of simplicity we extracted the most telling portion of this benchmark by taking 4 tables and 4 transactions, and keeping only the data accesses to these tables. Our subset models `customers` who each have a number of `accounts`. Each `account` is serviced by a particular stock `broker`, that in turn serves `accounts` from multiple `customers`). Each `customer` makes stock trades as part of a specific `account`. Each table has a unique integer id, along with other attributes. This dataset demonstrates that lookup tables are applicable to traditional OLTP workloads. The schema and our lookup table based partitioning strategies are shown in Fig. 4.

The four operations included in our TPC-E inspired benchmark are the following, as summarized in Table II.

*Trade Order:* A customer places buy or sell orders. This accesses the account and corresponding customer and broker records, and inserts a trade.

*Trade Result:* The market returns the result of a buy or sell order, by fetching and modifying a trade record, with the corresponding account, customer, and broker records.

*Trade Status:* A customer wishes to review the 50 most recent trades for a given account. This also accesses the corresponding account, customer and broker records.

*Customer Position:* A customer wishes to review their current accounts. This accesses all of the customer's accounts, and the corresponding customer and broker records.

This simple subset of TPC-E has three interesting characteristics: i) the `account` table represent a many-to-many relationship among `customers` and `brokers`, and ii) the `trade` is accessed both by `tid` and by `aid`, and iii) is rather write-intensive with almost 40% of the transactions inserting or modifying data.

To partition this application using hashing or ranges, we partition each table by id, as most of the accesses are via this attribute. The exception is the trade table, which we partition by account, so the trades for a given account are co-located. With ranges, this ends up with excellent locality, as TPC-E scales by generating chunks of 1000 customers, 5000 accounts, and 10 brokers as a single unit, with no relations outside this unit. This locality, however, is mostly a by-product of this synthetic data generation, and it is not reasonable to expect this to hold on a real application. As with Twitter, a multi-attribute partitioning cannot help with this workload. The account table most commonly accessed account id, but sometimes also by customer id. Using a multi-attribute partitioning would make these operations more expensive.

The partitioning using lookup tables for this dataset is obtained by applying the Schism partitioner [1] and therefore does not rely on this synthetic locality. The result is a fine-grained partitioning that requires lookup tables in order to carefully co-locate customers and brokers. Due to the nature of the schema (many-to-many relationship) we cannot enforce location dependencies for every pair of tables. However, thanks to the flexibility of lookup tables, we can make sure that the majority of transactions will find all of the tuples they need in a single partition. Correctness is guaranteed by falling back to a distributed plan whenever necessary.

The partitioning is shown in Fig. 4: customer and account are partitioned together by customer id. We also build a lookup table on `account.aid` and force the co-location of trades on that attribute. Brokers are partitioned by `bid`, and the partitioner tries to co-locate them with most of their customers. We also build a lookup table on `trade.tid`, so that the queries accessing trades either by `aid` or `tid` can be directed to a single partition. Similarly we add a lookup table on `customer.taxid`.

As shown in Table II, the lookup table implementation avoids broadcasts and two-phase commits, making all transactions execute on a single-partition. This results in much better scalability, yielding a 2.6× performance increase versus the best non-lookup table partitioning on our 10 machine experiments (we describe this experiment in detail in Section VI).

Given these examples for how lookup tables can be used, we now discuss several implementation strategies that allow us to scale lookup tables to very large datasets.

### B. Cost of Distributed Queries

Since the primary benefit of lookup tables is to reduce the number of distributed queries and transactions, we begin by examining the cost of distributed queries via a simple synthetic workload. We created a simple key/value table with 20,000 tuples, composed of an integer primary key and a string of 200 bytes of data. This workload is fits purely in main memory, like many OLTP and web workloads today. Our workload consists of auto-commit queries that select a set of 80 tuples using an id list. This query could represent requesting information for all of a user's friends in a social networking application. Each tuple id is selected with uniform random probability. We scale the number of backends by dividing the tuples evenly across each backend, and vary the fraction of distributed queries by either selecting all 80 tuples from one partition or selecting $80/k$ tuples from $k$ partitions. When a query accesses multiple backends, these requests are sent in parallel to all backends. For this experiment, we used 200 concurrent clients, where each client sends a single request at a time.

The throughput for this experiment with 1, 4 and 8 backends as the percentage of distributed queries is increased is shown in Fig. 6. The baseline throughput for the workload when all the data is stored in single backend is shown by the solid black line at the bottom, at 2,161 transactions per second. Ideally, we would get 8× the throughput of a single machine with one eighth of the data, which is shown as the dash line at the top of the figure, at 18,247 transactions per second. Our implementation gets very close to this ideal, obtaining 94% of the linear scale-out with 0% distributed queries.

As the percentage of distributed queries increases, the throughput decreases, approaching the performance for a single backend. The reason is that for this workload, the communication overhead for each query is a significant cost. Thus, the difference between a query with 80 lookups (a single backend) and a query with 20 lookups (distributed across 8 backends) is very minimal. Thus, if the queries are all local, we get nearly linear scalability, with 8 machines producing 7.9× the throughput. However, if the queries are distributed, we get a very poor performance improvement, with 8 machines only yielding a 1.7× improvement at 100% distributed queries. Therefore, it is very important to carefully partition data so queries go to as few machines as possible.

To better understand the cost of distributed queries, we used the 4 machine partitioning from the previous experiment, and varied the number of backends in a distributed transaction. When generating a distributed transaction, we selected the tuples from 2, 3 or 4 backends, selected uniformly at random.

The throughput with these configurations is shown in Fig. 7. This figure shows that reducing the number of participants in the query improves throughput. Reducing the number of participants from 4 backends to 2 backends increases the throughput by 1.38× in the 100% distributed case. However, it is important to note that there is still a significant cost to these distributed queries. Even with just 2 participants, the throughput is a little less than double the one machine throughput. This shows that distributed transactions still impose a large penalty, even with few participants. This is because they incur the additional cost of more messages for two-phase commit. In this case, the query is read-only, so the two-phase commit only requires one additional round of messages because we employ the standard read-only 2PC optimization. However, that means the 2 participant query requires a total of 4 messages, versus 1 message for the single participant query. These results imply that multi-attribute partitioning, which can reduce the participants in a distributed query from all partitions to a subset (e.g., $\sqrt{k}$ for a two-attribute partitioning), will improve performance. However, this is equivalent to moving from the "4/4 backends" line on Fig. 7 to the "2/4 backends" line. This improvement is far less than the improvement from avoiding distributed transactions. Since multi-attribute partitioning will only provide a modest benefit, and because it is not widely

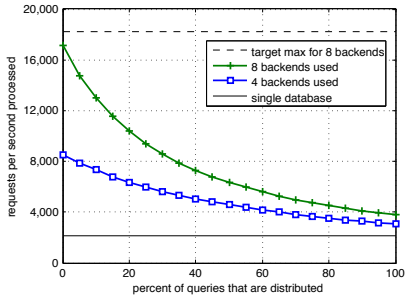


Fig. 6. Microbenchmark throughput with varying fraction of distributed queries

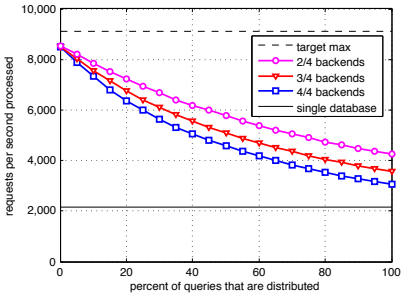


Fig. 7. Microbenchmark throughput with varying distributed query participants

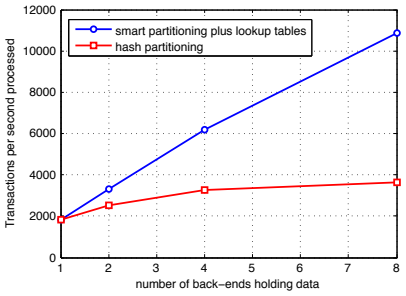


Fig. 8. Wikipedia throughput with varying backends

used today, we do not consider it further.

In conclusion, this microbenchmark shows that distributed queries and distributed transactions are expensive in OLTP and web workloads, where the cost of network communication is significant compared to the cost of processing the queries. Thus, lookup tables can help improve performance by reducing the number of participants in queries and transactions.

### C. Wikipedia

For this experiment, we used the Wikipedia dataset described in Section VI-A and partitioned it across 1, 2, 4 and 8 backends. The throughput in this case is shown in Fig. 8, for both hash partitioning and lookup tables. For lookup tables, we obtain $5.9\times$ higher throughput with 8 backends. While this is less than the ideal $8\times$ improvement, the trend on the graph is still mostly linear, implying that we could improve the throughput further by adding more backends. Using our implementation of lookup tables, described in Section VI-A, each query and transaction goes to a single partition. Thus, we would expect to see very close to the ideal scalability. In examining the CPU consumption across each backend, we observed that it varied across the backends in a very predictable way, with some servers always very close to 100%, while others were around 60-70%. The cause is that we are replaying a trace of web requests as recorded by Wikipedia, and so the requests are not balanced perfectly across the partitions over time. Thus, this imbalance means that we are not able to get perfect scalability.

In this experiment, hash partitioning does not obtain much improvement. With eight machines, we only get $2\times$ higher throughput compared to a single database with all data, which means that lookup tables performs $3\times$ better than hashing. The reason for this difference is that with hash partitioning, some of the queries (those that perform lookups by name) must be broadcast to all partitions. These queries limit the scalability. This example shows that in cases where a table is accessed by more than one attribute, lookup tables can drastically reduce the number of distributed queries that are required.

### D. Twitter

Our Twitter workload provides an example of how lookup tables can help by clustering many-to-many relationships together. As described in Section VI-A, we partitioned the Twitter data into 10 partitions using 1) hash partitioning and 2) three variants of lookup tables. The three variants are our own simple heuristic partitioning scheme, one-hop replication [4],

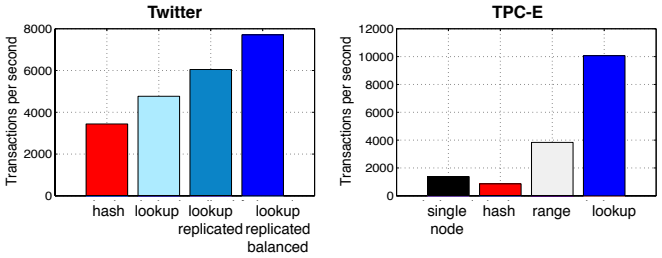


Fig. 9. Throughput improvement for Twitter and TPC-E datasets

and load-balanced one-hop replication. The throughput for these four configurations is shown on the left side of Fig. 9. In this case, our heuristic lookup table partitioning improves throughput by up to 39%. While this is only a modest improvement, this is purely due to the reduction in distributed queries by clustering users and replicating popular users. One-hop replication, because it makes every query local, improves the performance by 76% over simple hash partitioning. Finally, adding our load balancing algorithm yields a $2.24\times$ improvement over hash partitioning. All of these optimizations were possible because the lookup tables give us the flexibility to explore the placement of tuples in different ways, in an application-specific manner.

### E. TPC-E

For TPC-E, we partitioned a portion of the TPC-E benchmark across 10 machines as described in Section VI-A. The throughput for a single machine, hash partitioning, range partitioning, and lookup tables are shown in the right side of Fig. 9. Hash partitioning performs worse than a single database because of a large number of distributed deadlocks. Our implementation does not perform distributed deadlock detection, and instead relies on lock requests timing out after 5 seconds. This causes many transactions to stall and then restart, wasting resources. In this case, the CPUs on all the backends are mostly idle. The performance of this case could be improved through better deadlock detection, but this shows one of the hazards of distributed transactions. Range partitioning performs significantly better, despite the fact that it broadcasts the same queries that hash partitioning does. The reason is that the data accessed by a given transaction for range partitioning tends to be on a single backend. Thus in this case, all lock conflicts and deadlocks occur locally, and can be resolved quickly. Range partitioning across 10 machines yields a $2.78\times$ throughput improvement over a single machine.

Lookup tables allow most queries and transactions to go to

a single backend, due to their ability to direct queries based on more than one attribute. Due to this reduction in distributed queries and transactions, lookup tables provide $7.30\times$ better throughput than a single machine, and $2.6\times$ better than range partitioning. Lookup tables are not able to provide the ideal $10\times$ speedup because the workload still contains occasional queries that must go to every partition.

## VII. Related Work

Lookup tables can be used anywhere that partitioning data is important. Shared nothing distributed databases, from early work on R* [11], Gamma [12], and Bubba [2], to modern commercial systems like IBM DB2's Data Partitioning Feature rely on data partitioning for scalability. These systems typically only support hash or range partitioning of the data. However, even shared disk systems can use partitioning to optimize locality. Lookup tables can be used with all these systems, in conjunction with their existing support for partitioning.

In this paper, we argued that one use for lookup tables are as a type of secondary index for tables that are accessed via more than one attribute. Many alternatives to this strategy have been proposed. For example, Bubba proposed what they called Extended Range Declustering, where a secondary index on the non-partitioned attributes is created and distributed across the database nodes [2]. This permits more efficient queries on the secondary attributes, but still two network round trips: one to query the secondary index, then one to fetch the actual data. Our approach simply stores this secondary data in memory across all query routers, avoiding an additional round trip. MAGIC declustering is an alternative strategy, where the table is partitioned in a $N$-dimensional fashion using $N$ indexed attributes [3]. This ensures that all queries on the indexed attributes go to a subset of the nodes, but each subset will still need to be larger than one. In our approach, we can make queries via each attribute go to exactly one partition.

Previous work has argued that hard to partition applications containing many-to-many relationships, such as social networks, can be partitioned effectively by allowing tuples to be placed in partitions based on their relationships. Schism uses graph partitioning algorithms to derive the partitioning [1], while SPAR uses an online heuristic to rebalance partitions [4]. The Schism work does not discuss how to use the fine-grained partitioning it produces (the focus in that paper is on using classification techniques to transform fine grained partitioning into range partitions), and SPAR uses a distributed hash table to lookup up the tuple location, which introduces a network round-trip. Both these partitioning heuristics could use lookup tables to distribute queries.

An alternative approach is to force the user to use a schema and queries that are easily partitionable via a single attribute. The H-Store system is designed with the assumption that most OLTP applications are easily partitionable in this fashion [13]. Microsoft's SQL Azure [14] and Google Megastore [15] both force applications to execute only single partition transactions, providing mechanisms to specify how tables are partitioned together. This design removes the need for lookup tables, but also constrains how developers can use the database.

## VIII. Conclusion

We presented *lookup tables* as a way to provide fine-grained partitioning for transactional database applications. Using lookup tables, application developers can implement any partitioning scheme they desire, and can also create partition indexes that make it possible to efficiently route queries to just the partitions they need to access. We presented a set of techniques to efficiently store and compress lookup tables, and to manage updates, inserts, and deletes to them. We used lookup tables to partition three difficult-to-partition applications: Twitter, TPC-E, and Wikipedia. Each involves either many-to-many relationships or queries that access data using different keys (e.g., article ids and titles in Wikipedia.) On these applications, we showed that lookup tables with an appropriate partitioning scheme can achieve from 40% to 300% better performance than either hash or range partitioning, and shows greater potential for further scale-out.

## Acknowledgements

This work was supported by Quanta Computer as a part of the T-Party Project, and by NSF IIS-III Grant 1065219.

## References

[1] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning." in *VLDB*, Singapore, Sep. 2010.

[2] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez, "Prototyping Bubba, a highly parallel database system," *IEEE Trans. on Knowl. and Data Eng.*, vol. 2, no. 1, pp. 4–24, 1990.

[3] S. Ghandeharizadeh and D. DeWitt, "MAGIC: a multiattribute declustering mechanism for multiprocessor database machines," *IEEE Trans. Parallel Distrib. Syst*, vol. 5, no. 5, pp. 509 –524, May 1994.

[4] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez, "The little engine(s) that could: scaling online social networks," in *SIGCOMM*, New Delhi, India, Sep. 2010.

[5] G. Graefe, "Query evaluation techniques for large databases," *ACM Comput. Surv.*, vol. 25, pp. 73–169, Jun. 1993.

[6] P. Hunt, M. Konar, F. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems," in *USENIX*, Jun. 2010.

[7] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Elsevier Computer Networks*, vol. 53, no. 11, pp. 1830–1845, Jul. 2009. [Online]. Available: http://www.globule.org/publi/WWADH_comnet2009.html

[8] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi, "Measuring user influence in Twitter: The million follower fallacy," in *International Conference on Weblogs and Social Media (ICWSM)*, Washington DC, USA, May 2010.

[9] N. Kallen, "Big data in real-time at Twitter," in *QCon*, San Francisco, USA, Nov. 2010, http://qconsf.com/dl/qcon-sanfran-2010/slides/NickKallen_DataArchitectureAtTwitterScale.pdf.

[10] "TPC benchmark E," Transaction Processing Performance Council, Tech. Rep., Jun. 2010, version 1.12.0.

[11] C. Mohan, B. Lindsay, and R. Obermarck, "Transaction management in the R* distributed database management system," *ACM Trans. Database Syst.*, vol. 11, no. 4, pp. 378–396, 1986.

[12] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen, "The Gamma database machine project," *IEEE Trans. on Knowl. and Data Eng.*, vol. 2, no. 1, pp. 44–62, 1990.

[13] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an architectural era: (it's time for a complete rewrite)," in *VLDB*, Vienna, Austria, Sep. 2007.

[14] P. A. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. B. Lomet, R. Manne, L. Novik, and T. Talius, in *ICDE*, Apr. 2011.

[15] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *CIDR*, Monterey, CA, USA, Jan. 2011.