A Relational Database Schema on the Transactional Key-Value Store Scalaris

Nico Kruber, Florian Schintke (Zuse Institute Berlin)
{kruber, schintke}@zib.de

Michael Berlin
mail@michael.berlin

***Abstract*—Distributed key-value stores are horizontally scalable by design. However, structured data with links between values may raise hotspots or bottlenecks caused by popular keys and large index objects. These hotspots typically reduce the scalability of the key-value store, especially for operations changing data. Relational database management systems, on the other hand, are designed to handle relational data efficiently, but generally do not scale horizontally in a cost-efficient way. Combining the best of both worlds, would be great.**

With a wiki as a demonstrator, we map a relational database schema to a distributed transactional key-value store. This includes solutions for typical constraints key-value stores impose on applications due to their limited query expressibility. It also includes the mapping of dependent tables and secondary indices to a single key-value namespace. We evaluate and identify hotspots and bottlenecks and propose improved mappings. We reduce the effects of the most prominent hotspots, i.e. secondary indices, by applying advanced partitioning schemes which both reduce the size of the indices and allow more concurrent write accesses in transactional contexts. These optimisations are generic and help to map relational schemas and corresponding applications to transactional key-value stores in a way to preserve their horizontal scalability.

With our data models for key-value stores, we get the best of two worlds for the wiki application: a horizontally scalable database serving a moderately complex relational schema. Our optimisations give up to 96 % fewer transaction aborts for data change operations and an up to 25-fold latency improvement for the overall operations mix, i.e. reading, changing, and creating data, compared to the basic mapping, when replaying an access trace of the Wikipedia on our system.

***Index Terms*—scalable data model; relational schema; key-value store; P2P; DHT; Wikipedia; horizontal scalability**

I. Introduction

Relational databases are well prepared to handle inter-linked objects and are optimised for good vertical scalability. Their primary design goal is neither to be horizontally scalable nor to be distributed. In contrast, distributed key-value (*KV*) stores are optimised for independent objects and horizontal scalability.

We map relational database schemas to a flat key-value namespace of a distributed, transactional key-value store (Scalaris [1] in particular). We thereby want to maintain both: the horizontal scalability of the underlying key-value store and good performance for queries to inter-linked objects. To efficiently support complex queries, we show how to implement secondary indices and alternatives to the partial, ordered traversal of database tables on top of distributed key-value stores. Join queries in general cannot be supported efficiently in a distributed setting and are therefore beyond the scope of this paper.

This paper demonstrates the challenges, pitfalls, and side-effects which arise when implementing a relational schema on top of a scalable, transactional key-value store. Our running example throughout the paper is the mapping of a part of the Wikipedia database schema to a key value store. The proposed solutions, however, are applicable to other relational schemas as well, since the scenarios we address (e.g. hot entries; seldomly updated but frequently read items; selection from long, narrow tables; etc.) can also be found in many other practical database schemas.

- We map a relational database schema to a single key-value namespace. As a sample application, we map a subset of the MediaWiki SQL scheme[1] and deploy it on Scalaris [1] (Section III).
- For the most frequent queries, e.g. on indices, we identify hotspots and bottlenecks and propose optimised mappings using partitioning schemes and advanced access techniques to improve the scalability (Section IV).
- We study the influence of the data models on the performance and scalability of an exemplary Wiki-Servlet by re-playing actual Wikipedia traces (Section VI).
- We study the lock contention on index objects with different mapping approaches and observe up to 96 % fewer transaction aborts compared to the basic mapping due to more fine grained locking (Section VI).

Compared to our basic key-value data model, wiki page edit operations show an up to 20-fold improvement in latency and an increased scalability with the partitioned data models. Note, that we focus on the relative performance gains one can get from the conceptually different data models in this paper. We are not interested in tuning absolute performance or in comparing absolute performance to related systems here.

II. Background

Each node in a distributed key-value store is responsible for a partition of the key-space, i.e. a consecutive sub-range of the numerical domain keys and nodes are uniformly hashed to. Routing and ring-maintenance algorithms like Chord [2] or Chord$^{\#}$ [3] allow to find the responsible node for a given key in $O(\log n)$ network hops, when n is the number of nodes, and maintain the overlay structure on churn. The routing structure can be seen as a single, distributed, primary lookup index for the key-value store.

Our Scalaris [1] system[2]—in contrast to most other key-value stores—implements strongly consistent replication, quorum based fault-tolerance, and a fault-tolerant transaction protocol [4], which allows atomic updates on multiple key-value pairs. So, database-like applications can be developed on top of Scalaris.

A popular database schema, which we use as an example, is that of MediaWiki (Wikipedia). Figure 1 shows the core of the schema. Each wiki page has a unique title and some additional metadata. All page revisions are stored. The page object always refers to the latest revision. Pages are inter-linked by hyperlinks and overlapping category memberships.

[1] https://www.mediawiki.org/wiki/Manual:Database_layout
[2] http://scalaris.googlecode.com/

978-1-4799-5666-1/14/$31.00 ©2014 IEEE

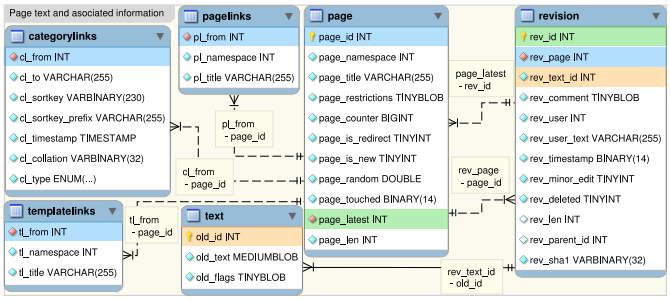

Figure 1. Core of Wikipedia's relational database schema[1].

III. Relational Database Schema in a KV Store

One way to map a relational database schema to a key-value store is to store all objects in a flattened namespace: The content of each row of a table is stored as a value under a key constructed from the original (unique) table name and the value of the primary-key of that table. We call this approach the *simple mapping*. For example, the metadata of a wiki page `x` is then stored as a tuple at key `page:x` or `x :page`. Objects are evenly distributed in the KV store and the routing leads to fast single-object access. With this approach, the following operations can already be served efficiently:

1) get the page metadata for a certain page title
2) get the current or a certain revision of a page
3) create a new page or a new revision of a page
4) check the existence of a page
5) get or update a statistical value

These operations involve a limited set of keys with medium-sized values and no overlap between operations on different pages. However, we also want to support the following operations efficiently:

6) get a list of all revisions of a page (history)
7) get all pages in a certain category
8) get all pages linking to a certain page (back-links)
9) get a random page

In the relational model these operations (partly) traverse the content of a table or rely on a secondary index. But table traversal in key-value stores would typically require lookup sequences with a logarithmic number of network hops in each step and secondary indices are not available.

In the following Sections III-A to III-D, we describe techniques for more elaborate, use-case dependent mappings which take frequently performed operations on the data and their interrelations as well as key-value store characteristics into account. Section III-E then outlines the steps that are necessary for page creation and page edits in our model.

A. Avoid Indirections to Other Objects

Network connections, wider geographic expansion, and logarithmic routing in the overlay result in much higher access latencies in a distributed system compared to a local one. Replication and fault-tolerance algorithms further increase the required effort of each operation. Therefore, indirections to other objects should be avoided in a key-value data model. To do so, we include part of the information of other objects into the primarily accessed one or fuse tables of the relational schema. There is a trade-off between improved efficiency for certain operations and the implied overhead to keep redundant data from such denormalisation [5] up to date.

The `page` table in the relational model, for example, stores metadata about pages, including a reference to the latest revision in the `revision` table, which then refers to the actual wiki text in the `text` table. Each page access thus has to retrieve data from three tables in sequence.

We eliminate the indirection to the wiki text by putting the (compressed) revision text inside its revision object and store the metadata of a page `x` in an object at the key `x:page`. Each revision (row) is stored at the key `x:rev:<id>` using a (page-)unique id. The latest revision object, however, is embedded into the corresponding page metadata object. This way, the most popular page query—retrieving the latest revision—can be performed with a single request.

B. Aggregate Objects to Avoid Table Traversals

By using secondary indices, a relational database gains fast access to objects matching some constraints. A key-value store, however, has no means to perform such a query. To overcome this in the simple mapping, each object could be linked to the next one and sequential traversal could be used.

The alternative is to explicitly materialise the secondary index by creating an object with all keys the index contains (Section III-C) or by aggregating all necessary data for a given query into an object and maintaining it consistently. If this aggregate object is more frequently read than the original objects are changed, there are good chances to get a benefit in terms of latency and throughput.

For example, to show the revision history overview (operation 6 on the left) an SQL query simply scans through its indices and retrieves the data for the web page from the `revision` table.

We create an aggregate object containing a list of all metadata needed for the history web page of page `x`, i.e. revision ID, timestamp, minor edit flag, contributing user, comment, and revision size and store it at key `x:revs`.

C. Index Objects to Avoid Table Traversals

Similarly to the use of aggregate objects from the previous section, we can create application level index objects for each domain of a secondary index in an RDBMS. Without loss of generality we represent such indices as lists. For example, we store a list of all primary keys of a table to be able to traverse through and filter among them in the application.

An example for index objects is the request for showing a random page (operation 9 on the left). In MediaWiki, it is a two-phase operation: first a random page title is selected by scanning through the index on the `page_random` column, then an HTTP redirect for the selected page is sent back to the user. In order to select a random page with a key-value store, we create an index object containing all page titles (page list).

We also use application level index objects to support back-links, i.e. the inverse direction of a link, category memberships, or template use. In the relational model these back-links are expressed in link tables (e.g. `templatelinks` in the Wikipedia schema). These tables store m:n relations, e.g. each page can use m templates and each template can be used by n source pages.

For our wiki application to render a back-link page we only need its n sources (operations 7 and 8 on the left) and thus store page title lists for each back-link page `x` at `x:blpages`, `x:cpages`, and `x:tpages`, respectively.

D. Application Level Caching

For certain operations, like showing statistical information or visual feedback with no consistency constraints, the timeliness is not that relevant and data can be cached at the application level.

For example, checking the existence of a page (operation 4 on the left) can be realised in three ways: (a) trying to read the according page object, (b) retrieving the page list and checking there, or (c) using a cache of the (regularly updated) page list. During a page retrieval we will use the first method as we need the page content anyway (if it exists). For visual feedback whether a link's target page exists (normal link) or not (link in red) we use the cache, as having a single request for each such link would be too expensive.

Our overall *basic key-value data model* is shown in Table I. An additional page list counter allows for quick access to the number of pages sparing a full index retrieval.

Table I
RELATIONAL WIKI MODEL AS A KEY-VALUE SCHEMA

`<title>:page`	page object (incl. current revision)
`<title>:rev:<id>`	any other revision of a page
`<title>:revs`	metadata for all revisions of a page
`<title>:cpages`	pages in this category
`<title>:blpages`	pages linking here
`<title>:tpages`	pages using this template
`pages`, `pages:count`	list of all page titles and its counter
`stats:pageedits`	total number of page edits

E. Page Creation and Edits in Our Basic KV Model

Creating and editing pages (operation 3 in the beginning of Section III) does not only involve saving its contents and metadata. We also need to update all affected indices with the new information. We thus include all of the following tasks in a single transaction:

1) store the new revision
2) create or update the page object
3) add new or remove old back-links, templates and categories
4) if a new page is created: add an entry to the page list and increase page counter(s)

This transaction may become big depending on the actual payload of the edit operation. Especially task 3 can create a lot of entries in the transaction since for every new or removed connection the target's back-link index needs to be updated with the edited page's title. Note that a fifth step updates the edit stats counter outside of this transaction as a trade-off between performance and consistency. The latter is not necessarily required from a statistical value.

F. Concurrency Limitations by Large Index Objects

For page read operations almost all requests execute a single read for the page object and are independent from each other. Edit and create operations, though, use transactions involving multiple keys which are only independent from each other if index objects are unchanged. If index objects change, transactions become bigger and may overlap and form an (aborted) *concurrent edit*. The larger the overlap and transaction runtime, the higher the probability of a concurrent edit.

IV. PARTITIONING APPLICATION LEVEL INDICES

Application level index objects may become hotspots if they are accessed more frequently than other objects. The effects of these hotspots vary depending on the scale of the hotspot: a *global* hotspot includes a very broad group of objects, e.g. the list of all page titles, and thus has a big influence on the scalability. A *local* hotspot on the other hand only connects small groups of objects, e.g. the revision history index, with a limited influence on the overall scalability.

Global or local hotspot, both limit the scalability for the same reasons: (a) size and (b) concurrent accesses, especially in case of write operations in transactional contexts. Concurrent accesses to an index are not possible—only one process can change an object at a time. The larger the objects are, the longer it takes to execute the write transaction and the lower is the overall transaction throughput.

By having operations on hotspots inside a write transaction, we limit the transaction's concurrency as well. Formerly independent operations become dependent via the index objects.

A. Simple Partitioning Schemes for Hotspots

To reduce the effects of hotspots we split indices or global counters into *partitions* reducing their size and improving the write concurrency. We consider two methods to spread objects to partitions: (a) *randomly* (mainly for counter values) and (b) *hash-based* (to split large lists) (see Figures 2 and 3).

For each index we have to choose how many partitions we want. When the index grows too much or the application needs to support more writers, a re-partitioning with a higher number of partitions may be needed to maintain scalability.

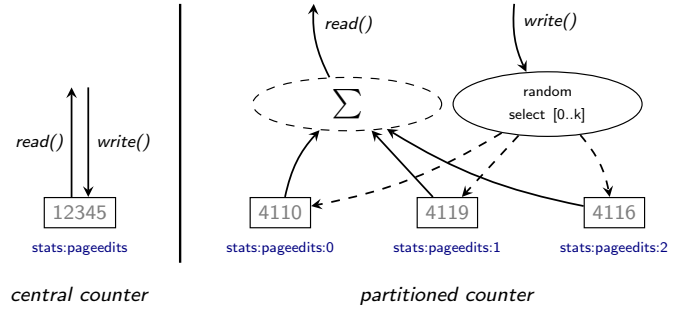

Figure 2. Spreading write load on counter objects by partitioning and random object selection.

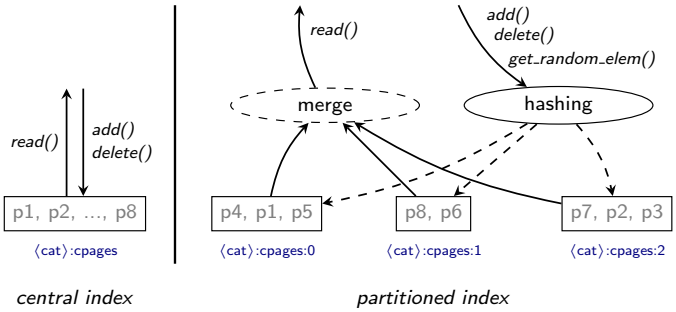

Figure 3. Partitioning an index with placement via hashing.

Suppose we choose to have k partitions. Performance is improved in two aspects: (a) *big objects* are roughly reduced in size by a factor of k and (b) possible *concurrent write accesses* are increased by a factor of k. Both positively affect accesses to a single partition, e.g. write transactions or the `get_random_elem()` operation in Figure 3. However, the performance of reading the whole value, e.g. to retrieve all members of a wiki category, decreases with growing k.

B. Advanced Partitioning Schemes for Hotspots

To decouple the scalability of write transactions (size and concurrency) from the scalability of read operations (size and number of partitions), we propose a hash-based partitioning scheme with different sets of *read* and *delta buckets* (see Figure 4) similar to the concept of Log-Structured Merge (LSM) Trees [6]. During normal operation the application does not change the read buckets. Instead, the (initially empty and generally small) *delta buckets* log additions, modifications, and deletions of items. Periodically, the delta buckets may be merged back into the read buckets and can then be cleared.

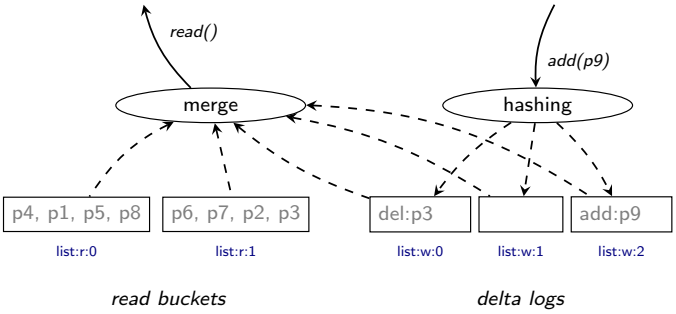

Figure 4. Partitions with hashing and read/delta buckets.

This allows to independently tune the partition size (via read buckets) and write concurrency (via delta buckets). Since the delta buckets only contain a few number of items (if any), we also have a smaller payload for write transactions. Because write concurrency is not affected by reducing the number of read buckets, read-all operations become cheaper.

V. Related Work

Previous research on decentralised wiki systems [7], [8], [9], [10] focused mainly on individual page operations. Therefore, the scalability issues of index objects—which are necessary to provide advanced wiki features like page history, categories, or back-links—were not identified before.

Piki [8] is a pure peer-to-peer based wiki engine which supports concurrent editing, version control, and decentralised full-text search. Similarly to PeerCollaboration [11], it stores for each word a list of referencing pages. Such keys may become subject to contention for popular words similar to our index objects and may be resolved accordingly.

Urdaneta et al. [10] proposed a decentralised wiki engine focussing on the aspects of untrusted nodes' membership.

Other decentralised wiki systems mainly focus on support for disconnected operations [9] or concurrency-awareness to improve collaboration [12]. These solutions can be applied orthogonally to our approach (on the client side).

Plantikow et al. [13] leverage range queries and store each back-link as a separate key prefixed by the page name. A range query with this prefix can be used to determine all of them. With our index objects, however, we reduce the transaction size and do not require range query support.

As an alternative to application level index objects, the underlying data store could maintain secondary indices, like some document-oriented NoSQL stores, e.g. MongoDB, or column-family based NoSQL stores, e.g. Cassandra, do. These systems allow to define a schema for the stored values which enables the automatic translation of a subset of SQL queries, e.g. by the middleware presented in [14]. However, the definition of constraints on inter-linked objects, e.g. foreign keys in SQL, cannot be supported automatically and manual mapping optimisations and application logic using transactions remains necessary.

H-Store/VoltDB, Calvin, and NuoDB are transactional, relational and horizontally scalable data stores related to Scalaris. While we try to stick to the basic concept of a transactional key-value store, which is completely distributed, and then optimise its use in more complex usage scenarios, these systems try to distribute relational schemas directly, which leads to other solutions. Calvin, for example, puts a transaction layer on top of storage systems and requires complete server-side processing of the transactions.

Other approaches for (partly) automatic database partitioning and database elasticity were proposed in [15], [16], [17], [18], but do not focus on DHT based key-value stores.

Hyperdex Warp [19] is a system similar to Scalaris but uses other techniques to ensure fault-tolerance and ACID properties on transactions across several objects. Our techniques to avoid hotspots in the data store should be similarly applicable and beneficial to Hyperdex Warp because the observed hotspots still remain the same.

VI. Evaluation

For our experiments, we imported the Spanish Wikipedia dump from 1 Oct 2007 including only the latest revision of each of the ca. 700 000 pages. After the import into Scalaris 0.7.2 on Erlang R14B04, less than 8 GiB RAM are used thus also making this dump feasible for evaluations with small system sizes. Table II shows the distribution of the different kinds of objects of the Spanish Wikipedia dump in our KV-model and some statistics about their sizes. All benchmarks were executed on a 33-node cluster with dual quad-core AMD Opteron 2370, 8 GiB RAM, and GbE.

Table II
Quantity and sizes of objects in the basic KV-model, filled with a Spanish Wikipedia dump from 1 Oct 2007.

Type	Count	Size (in KiB)			
		avg	min	max	σ
page	714 763	1.5	0.4	428.4	3.0
revision list	714 763	0.4	0.3	0.6	0.0
pagelist	20	317.4	0.2	4 306.4	939.4
backlink pagelist	1 588 391	0.3	0.2	481.8	1.1
category pagelist	43 082	0.4	0.2	23.7	0.4
template pagelist	13 964	0.6	0.2	455.3	6.0
pagelist count	20	0.2	0.2	0.2	0.0
category count	43 082	0.2	0.2	0.2	0.0

Additionally to the *basic KV-model*, we choose partitioning schemes based on the actual value sizes and reasonable concurrency needs. The simple partitioning P_{simple} splits the page list index so that each partition is smaller than 100 KiB. The corresponding counters use the same scheme. The category, template, and back-link lists are each split into 5 partitions mainly for concurrency considerations but also for the size of some of these lists. While these partitions use hashed item placements, it is sufficient to use the random select strategy for the global edit stat counter, using 50 partitions. For $P_{rd-buckets}$ we only need to consider concurrency and thus split the page list into 6 read and 10 delta buckets. This setup has fewer partitions and thus cheaper read operations. However, the supported level of concurrency is lower than in P_{simple}. The category, template, and back-link lists use 1 read and 5 delta buckets and thus support the same level of concurrency as in P_{simple}. The edit stat counter uses the same scheme as in P_{simple}. A summary is provided by Table III.

Table III
Partitioning schemes P_{simple} and $P_{rd-buckets}$.

	$\mathbf{P_{simple}}$	$\mathbf{P_{rd-buckets}}$
page list index	$hash(50)$	$hash(6\,read, 10\,delta)$
page list counter	$hash(50)$	$hash(1\,read, 10\,delta)$
back-link lists	$hash(5)$	$hash(1\,read, 5\,delta)$
edit stat counter	$random(50)$	$random(50)$

A. Benchmark Setup

We used an extended version of WikiBench [20] to process an original Spanish Wikipedia trace [21] from October 2007 and replayed it with various rates on Tomcat 7.0.42. We studied several scenarios by filtering the original trace and created a typical *back-end scenario* with a 95 % reduced number of page read operations—a typical rate of reads that is served by an HTTP cache [21]. This scenario contains 97.2 % reads, 2.5 % edits, and 0.3 % creates.

The WikiBench controller process reading the trace file and distributing the work according to the configured scenario runs on a dedicated node. Four more nodes serve as WikiBench workers and simulate up to 140 clients per Scalaris & Wiki node. We can thus evaluate systems of up to 28 nodes and start with system sizes of 4 due to Scalaris always using 4 replicas for each object.

B. Wiki Operation Performance

The individual runtimes of reads, edits, and creates with the three data models are depicted in Figure 5 in a benchmark with no concurrency. Therefore, differences in the *Scalaris* runtimes are solely a result of reducing the indices' sizes with partitioning. This can be clearly seen with the create operation that—in contrast to an edit—always includes the global pagelist. The optimisations of the other indices can be seen in the edit operation results. Note that the

roughly 10 ms used by *Tomcat* include the time needed for parsing and rendering the wiki text using the Bliki library[3]. *Other* includes overhead from e.g. downloading the HTML page.

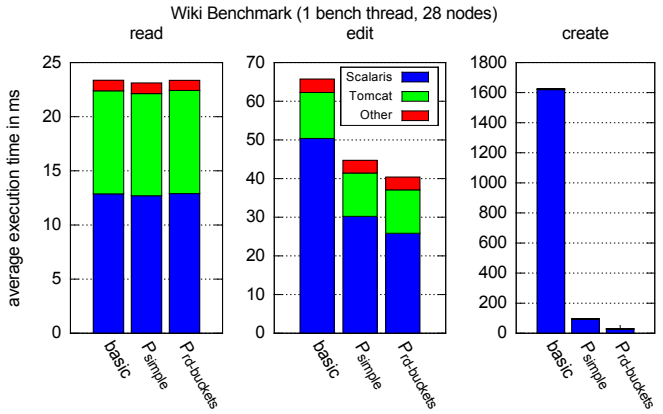

Figure 5. WikiBench HTTP request execution times per component.

To better understand the differences between these three operations, please note that each HTTP request may result in multiple requests to the Scalaris DB which may also involve multiple objects stored at different keys. Table IV shows the average number of Scalaris keys per HTTP request in the read-/edit-/create-only as well as the back-end scenario. During the rendering, read operations may issue additional requests for pages or statistics to include. These additional requests will be handled by separate Scalaris requests. Edit and create operations, however, need to update multiple backlink indices and do so inside a single request. The back-end scenario contains a mix of these three operations (ref. Section VI-A).

Table IV
AVERAGE NUMBER OF SCALARIS KEYS PER HTTP REQUEST (28 NODES, 1 BENCH THREAD)

	read[1]	**edit**[2]	**create**[3]	**back-end**[1]
basic	4.18	10.32	13.15	4.35
P_{simple}	4.37	10.32	13.15	4.57
$P_{rd-buckets}$	4.35	10.32	13.15	4.51

[1] 100 000 ops [2] 18 884 ops [3] 4 536 ops

More in-depth per-function execution times of a real benchmark with concurrency are shown in Figure 6 for some of the core functions, i.e. the edit (SAVE), page retrieval (PAGE), getting the current number of pages (PAGECOUNT), and getting a list of all pages in a category (CATLIST) operations.

The slowest operation in the basic KV-model, as expected, is the SAVE operation. This is because on average, 10-13 keys need to be written to Scalaris in a single transaction. The other operations shown in Figure 6, however, only read a single key-value object.

Since retrieving a single page object is identical in the three models, we assume that the high cost for SAVE operations causes the slow page retrievals as shown. Recall that we use non-blocking reads so there is no direct interrelation between the edit and read operations. The cost for SAVE operations is also the reason why the *basic* model is more overloaded than the partitioned data models at the same rate and other operations may not be compared directly (see Figures 7 and 8, e.g. at a 2 000 rate like here).

Figure 6, however, clearly indicates the penalties we got from partitioning the page counter. The 50 partitions of P_{simple} are more expensive than the $1 + 10$ buckets from $P_{rd-buckets}$ or the single key in *basic*. On the other hand, $P_{rd-buckets}$ needs to read $1 + 5$ buckets for the category list while P_{simple} has only 5 partitions. Both, however, support the same level of concurrency.

[3]Java Wikipedia API (Bliki engine): https://code.google.com/p/gwtwiki/

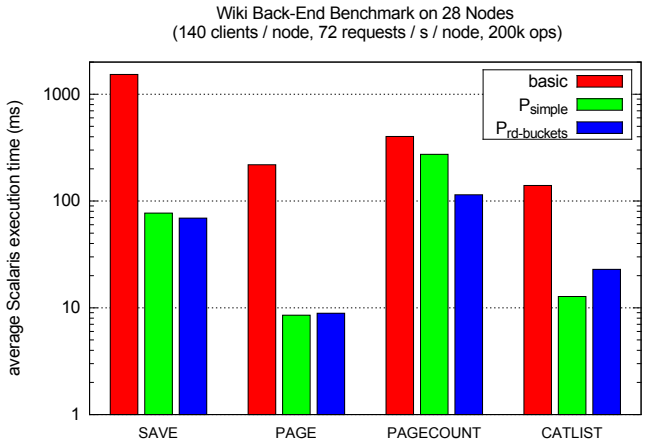

Figure 6. Scalaris execution times of selected operations.

C. Performance Characteristics with Varying Load

We now analyse benchmarks on a fixed-size system in different load situations using the back-end scenario. We focus on three aspects: (a) the achieved throughput compared to the requested rate (HTTP requests / s), (b) the latencies of HTTP operations, and (c) the transaction aborts due to concurrent edits/creates and the number of successful edit/create operations, respectively.

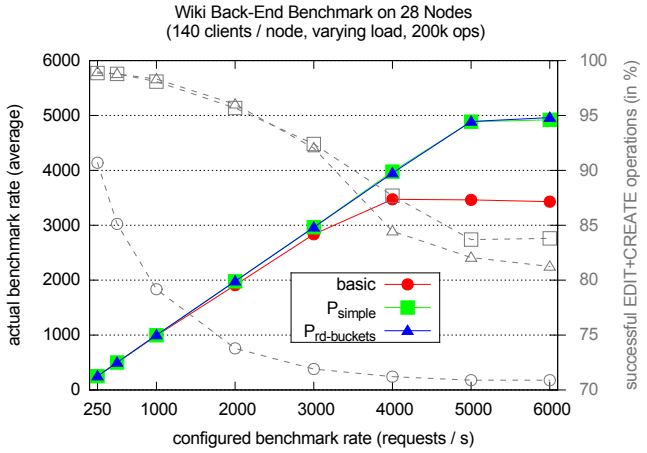

Figure 7. Wiki throughput with increasing load.

All three data models show similar behaviour in the throughput analysis (Figure 7) up to a rate of 3 000 with a slight advantage for the partitioned data models. These rates more and more saturate the system with a big difference in the number of successful edit and create operations where, as expected, P_{simple} and $P_{rd-buckets}$ allow more concurrency than *basic*. Please note that due to the lower number of page list partitions in $P_{rd-buckets}$ fewer create operations than in P_{simple} are successful in overloaded situations therefore lowering the overall number of successful operations (also ref. Figure 9). Rates above 3 000 lead to an overloaded system where the actual rate does not match the configured rate any more and the number of successful operations decreases at a higher rate (up to a point where the system does not allow higher rates).

More detailed insights are available by looking at the latencies of the HTTP requests (Figure 8). *basic* seems overloaded starting at a rate of 2 000 although it was able to sustain a 3 500 rate. The partitioned data models indicate a beginning overload situation at a rate of 3 000 and above. Looking at the highest latencies of

Table V
TOP 10 FAILED KEYS OF EDIT OPERATIONS IN THE BASIC KV MODEL (28 NODES, 2 000 $^{requests}/_{second}$)

# of Failures	Key
44	España:**blpages**
44	Category:Novelistas:**cpages**
44	Category:Novelistas:**cpages:count**
25	**articles:count**
22	Template:Usuario no reggaeton:**tpages**
21	Template:Usuario no reggaeton:**blpages**
19	2007:**blpages**
17	Argentina:**blpages**
16	User:No sé qué nick poner/Usuario no reggaeton:**blpages**
15	Template:Commons:**tpages**

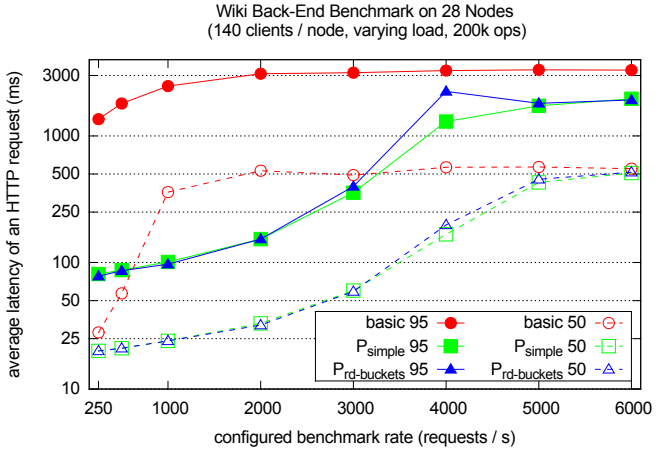

Figure 8. 95-/50-percentile latencies with increasing load.

the 95 % fastest requests (95-percentile), the differences between P_{simple}/$P_{rd-buckets}$ and *basic* in the rates below 3 000 are a factor of 16 to 25. The median (50-percentile) of these latencies varies by a factor of up to 17 with the highest difference at rate 2 000. The more the system is overloaded, the more the latencies of the different data models converge. However, Figure 7 shows that P_{simple} and $P_{rd-buckets}$ can sustain higher rates with these latencies than *basic*.

Note that because the (longer-lasting) edit and create operations only make up 2.1 % of all operations in this scenario, they are not even present in the 95- or 50-percentile. Their overhead, however, influences the remaining operations as shown.

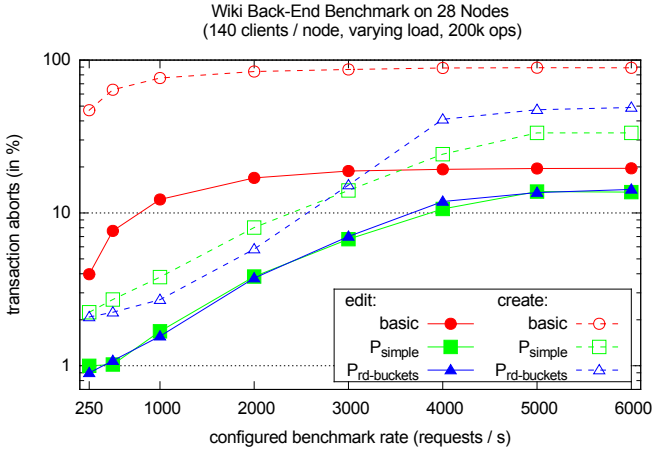

Figure 9. Wiki Edit and Create aborts with increasing load.

Figure 9 shows, in more detail, the number of transaction aborts of edit and create operations and thus the implications of having a data model with hotspots. The global page list effectively prevents concurrent creates in the *basic KV-model* despite their rare occurrences. Partitioned data models allow higher concurrency resulting in fewer aborts, e.g. only 4.1 % and 2.9 %, respectively, versus 75 % at a 1 000 rate. Edits do not modify the global page list and thus perform better. Nevertheless, P_{simple} and $P_{rd-buckets}$ only have 1.7 % and 1.4 % aborts, respectively, at a 1 000 rate vs. 12.7 % using the *basic KV-model*. In general, $P_{rd-buckets}$ is slightly better than P_{simple} except for create operations in scenarios with very high load. In these systems, the number of concurrent create operations increases above the level of concurrency the chosen partition sizes allow (ref. Table III). P_{simple} uses more partitions and thus has fewer aborts than $P_{rd-buckets}$.

Except for timeouts due to overload, edit and create operations may fail due to concurrent transactions trying to modify the same key. Table V shows the Top 10 of the keys which caused edit operations to fail—create operations mostly fail due to the page list keys and their counters. All Top 10 entries are occupied by application level indices thus confirming them as hotspots.

Generally, there are two types of edit operations which may lead to a failure: (a) concurrent edits of the same page, i.e. two users modifying the same page and (b) independent edits with an overlapping write set, e.g. due to secondary indices. Table VI shows the ratio of these different reasons for aborts.

Table VI
TRANSACTION ABORT TYPES OF EDIT OPERATIONS
(28 NODES, 2 000 $^{requests}/_{second}$)

	basic	$\mathbf{P_{simple}}$	$\mathbf{P_{rd-buckets}}$
concurrent same page edits	453	180	175
overlapping write set aborts	613	36	30
total aborts	1 066	216	205

Concurrent edits of the same page can only be resolved indirectly by a data model in our benchmarks by increasing the operations' performance and thus having fewer simultaneous edits. By using multiple partitions for secondary indices, we effectively reduce the chance of having an overlapping write set. Both effects are shown by Table VI: the performance improvements (ref. Figure 5) reduce the chance for concurrent edits of the same page by 40 %. Meanwhile, partitions allow higher levels of concurrency. Together, these optimisations reduce overlapping write set aborts by 95 %.

D. Read Scalability

As expected, the throughput of the read operations in a read-only scenario with maximum load (Figure 10) scales linearly using any of our KV-models with a good relative speed-up of 25.3 in the 28-node system. Note that there is a very small advantage for the basic model due to cheaper page list read operations.

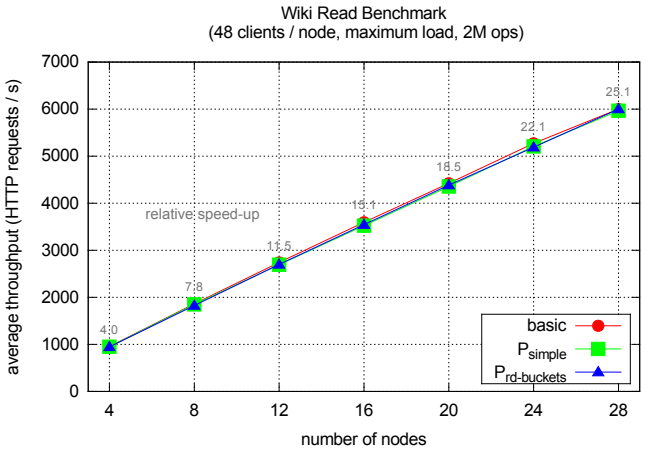

Figure 10. Wiki read-only scalability under maximum load.

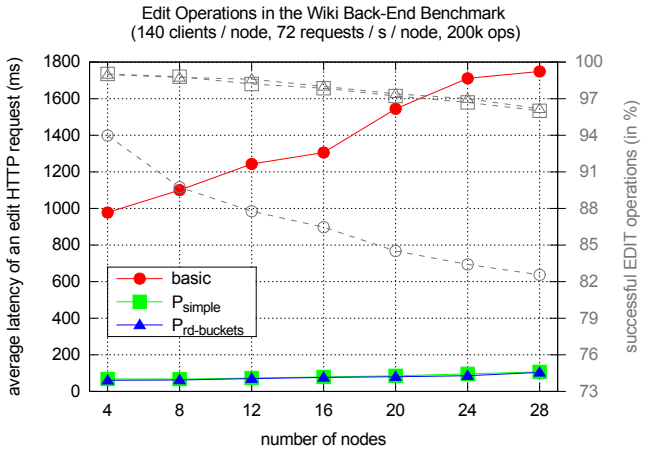

Figure 12. Back-end wiki edit scalability under high load.

E. Edit Scalability in the Back-End Scenario

As a basis for further evaluations of the more interesting edit operations in systems of different sizes, we will use experiments with the back-end scenario at two different loads based on the findings of Section VI-C:

- *high load*: 72 requests per second and node
- *overload*: 180 requests per second and node

On a 28-node system, high load roughly equals 2 000 requests per second and the overloaded scenario is similar to the 5 000 requests per second where the partitioned data models become overloaded.

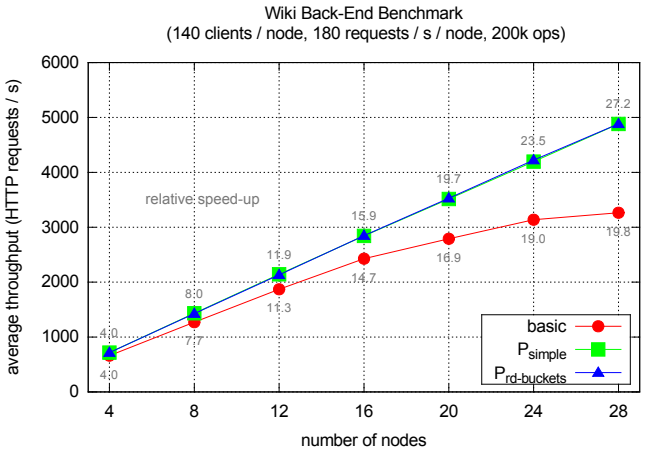

Figure 11. Wiki back-end scalability in an overload situation.

Figure 11 shows the throughput and the relative speed-ups of the three data models in the overloaded scenario. In contrast to the high load scenario where all models can sustain the rate (not shown here), *basic* is too overloaded. Since the rate is capped, however, we will focus on the—ca. 5 000—edit operations' latencies as well as the percentage of successful, i.e. not aborted, transactions. The throughput is only presented to put these results into perspective.

Figure 12 shows the results from all edit operations in the *high load* scenario. Although the load increases equally with the number of nodes, the latency of the *basic KV-model* increases by 79 % from 4 to 28 nodes. P_{simple} and $P_{rd-buckets}$ add 56 % and 70 %, respectively. In all setups $P_{rd-buckets}$ has a lower latency and therefore a higher number of successful operations than P_{simple}. It achieves a 16-fold (4 nodes) to 17-fold (28 nodes) improvement in latency over *basic* with a maximum at 24 nodes (20-fold). Nevertheless, despite our use of partitioned models the latency increases and the number of successful operations still decreases. Recall, though, that the concurrency in this setup increases linearly but the partition sizes always stay fixed. In order to scale to any concurrency, partition sizes need to be adapted dynamically to the need of the particular hotspots.

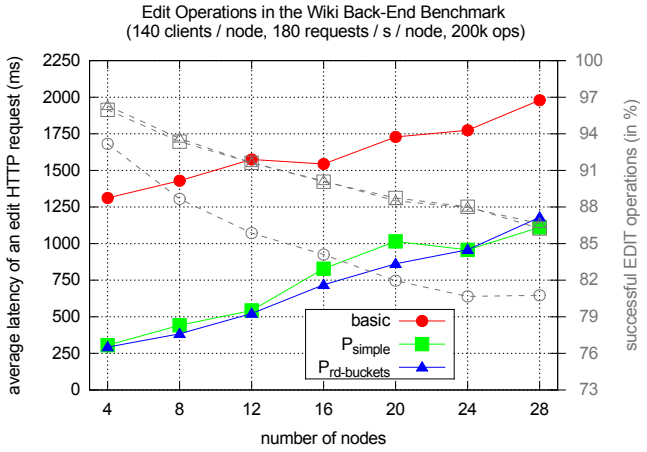

Figure 13. Back-end wiki edit scalability in an overload situation.

The results of the edit operations in the *overloaded* scenario are shown in Figure 13. In all three data models, the linearly increasing concurrency imposed by the overloaded scenario is too high for the fixed level of concurrency the models provide. The average latencies in P_{simple} and $P_{rd-buckets}$ differ from *basic* by a factor of 1.7 (28 nodes) to 4.5 (4 nodes). The number of successful operations in the *basic* model is up to 10 % lower than in P_{simple} or $P_{rd-buckets}$. All of them decrease though. Please note that the overloaded comparison between *basic* and the partitioned data models is a bit unfair. The rate *basic* achieves on systems with more than 16 nodes is considerably slower than the rate with the partitioned data models (ref. Figure 11). *Basic* is already saturated and thus its latencies and the number of failed edits do now increase any more.

F. Edit-only Scenarios

In order to see how the models behave in scenarios with a higher number of edit operations, we set up a system where only edit operations are performed. Figure 14 shows the latencies of the 28-node system in different load scenarios (HTTP requests per second). For *basic*, rates above 200 are too high and some requests

even time out (60 s) in which case the benchmark is aborted. Both partitioned models scale up to 700 requests per second. As indicated by the number of successful edit operations, rates above 700 lead to an overloaded system. Generally, $P_{rd-buckets}$ has more successful operations than P_{simple} and is also able to sustain a higher rate.

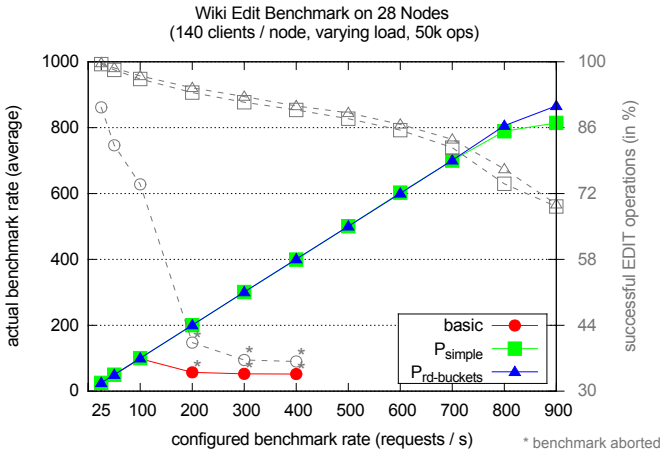

Figure 14. Edit operations with increasing load.

Consequently, in the maximum load scenario shown by Figure 15, $P_{rd-buckets}$ always allows higher throughputs than P_{simple} (except for the outlier at 8 nodes). In turn, the rate of successful operations between these two models is the same. Both scale with a relative speed-up of 16.6 and 17.0 in the 28-node system, respectively. The *basic* model, however, does not scale above 16 nodes and has an up to 17 % lower rate of successful operations.

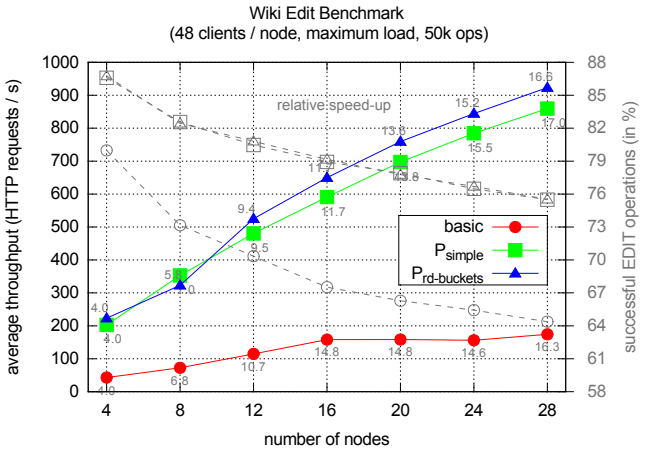

Figure 15. Edit scalability with maximum load.

VII. Conclusion

We mapped a relational database schema to the transactional key-value store Scalaris and proposed several generic techniques to implement the functionality of secondary indices. We also showed how to avoid hotspots and bottlenecks that would prevent scalability. In a practical evaluation based on replaying Wikipedia access traces at different rates we demonstrated the scalability of our approach.

Scalaris and the Wikipedia-clone on top of it, including a dump-loader that loads Wikipedia XML-dumps into Scalaris, can be found in our open source code repository at http://scalaris.googlecode.com. We used version 0.7.2 of Scalaris and the Wikipedia-clone for all evaluations.

While our findings actually came from the task of mapping a part of the Wikipedia schema to our key-value store, we are confident that they can help in other use cases. For example, they could be automated and applied dynamically in a self-optimising system. In such a scenario one would additionally need scalable online data layout migration techniques, cost models for data layout migrations, as well as distributed monitoring and steering components based on feedback loops and control theory, which would be far beyond the scope of this work.

Acknowledgments

We thank the rest of the Scalaris team for the key-value store implementation itself and their valuable support. We also thank Guillaume Pierre for making the Wikipedia traces [21] available. This work received funding from the EU FP7 projects *4CaaSt* (GA no 258862), *Contrail* (GA no 257438), and the EU CIP ICT-PSP project *IES Cities* (GA no 325097).

References

[1] T. Schütt, F. Schintke, and A. Reinefeld, "Scalaris: Reliable Transactional P2P Key/Value Store," in *ACM SIGPLAN Erlang Workshop*, Sep. 2008.

[2] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, Aug. 2001, pp. 149–160.

[3] T. Schütt, F. Schintke, and A. Reinefeld, "Range queries on structured overlay networks," *Computer Communications*, vol. 31, no. 2, pp. 280–291, Feb. 2008, special Issue: Foundation of Peer-to-Peer Computing.

[4] F. Schintke, A. Reinefeld, S. Haridi, and T. Schütt, "Enhanced Paxos Commit for Transactions on DHTs," in *CCGRID*. IEEE, May 2010, pp. 448–454.

[5] M. Schkolnick and P. Sorenson, "Denormalization: a performance oriented database design technique," in *AICA Congress, Bologna, Italy*, 1980.

[6] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[7] J. C. Morris, "DistriWiki: a distributed peer-to-peer wiki network," in *Proceedings of the 2007 international symposium on Wikis*. ACM, 2007, pp. 69–74.

[8] P. Mukherjee, C. Leng, and A. Schurr, "Piki-a peer-to-peer based wiki engine," in *Peer-to-Peer Computing, 2008. P2P'08. Eighth International Conference on*. IEEE, 2008, pp. 185–186.

[9] G. Canals, P. Molli, J. Maire, S. Laurière, E. Pacitti, and M. Tlili, "XWiki Concerto: A P2P wiki system supporting disconnected work," in *Cooperative Design, Visualization, and Engineering*. Springer, 2008, pp. 98–106.

[10] G. Urdaneta, G. Pierre, and M. Van Steen, "A Decentralized Wiki Engine for Collaborative Wikipedia Hosting." in *WEBIST (1)*, 2007, pp. 156–163.

[11] T. Bocek and B. Stiller, "PeerCollaboration," in *Scalability of Networks and Services*. Springer, 2009, pp. 183–186.

[12] A. Craig, A. Davoust, and B. Esfandiari, "A Distributed Wiki System Based on Peer-to-Peer File Sharing Principles," in *Proceedings of the IEEE/WIC/ACM WI-IAT '11*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 364–371.

[13] S. Plantikow, A. Reinefeld, and F. Schintke, "Transactions for Distributed Wikis on Structured Overlays," in *DSOM*, ser. LNCS, A. Clemm, L. Z. Granville, and R. Stadler, Eds., vol. 4785. Springer, Oct. 2007, pp. 256–267.

[14] J. Rith, P. S. Lehmayr, and K. Meyer-Wegener, "Speaking in tongues: SQL access to NoSQL systems," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, ser. SAC '14. New York, NY, USA: ACM, 2014, pp. 855–857.

[15] J. Tatemura, O. Po, and H. Hacigümüs, "Microsharding: a declarative approach to support elastic OLTP workloads," *Operating Systems Review*, vol. 46, no. 1, pp. 4–11, 2012.

[16] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," *PVLDB*, vol. 3, no. 1, pp. 48–57, 2010.

[17] R. V. Nehme and N. Bruno, "Automated partitioning design in parallel database systems," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, Eds. ACM, 2011, pp. 1137–1148.

[18] A. Pavlo, C. Curino, and S. B. Zdonik, "Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, Eds. ACM, 2012, pp. 61–72.

[19] R. Escriva, B. Wong, and E. G. Sirer, "Warp: Lightweight multi-key transactions for key-value stores," Cornell University, Ithaca, Tech. Rep., 2013.

[20] E.-J. van Baaren, "Wikibench: A distributed, Wikipedia based web application benchmark," Master's thesis, Vrije Universiteit Amsterdam, 2009.

[21] G. Urdaneta, G. Pierre, and M. Van Steen, "Wikipedia workload analysis for decentralized hosting," *Computer Networks*, vol. 53, no. 11, pp. 1830–1845, 2009.