Block as a Value for SQL over NoSQL

Yang Cao[1] Wenfei Fan[1,2,3] Tengfei Yuan[1]
[1]University of Edinburgh [2]Beihang University [3]SICS, Shenzhen University
{yang.cao@, wenfei@inf., tengfei.yuan@}ed.ac.uk

ABSTRACT

This paper presents Zidian, a middleware for key-value (KV) stores to speed up SQL query evaluation over NoSQL. As opposed to common practice that takes a tuple id or primary key as key and the entire tuple as value, Zidian proposes a block-as-a-value model BaaV. BaaV represents a relation as keyed blocks (k, B), where k is a key of a block (a set) B of partial tuples. We extend relational algebra to BaaV.

We show that under BaaV, Zidian substantially reduces data access and communication cost. We provide characterizations (sufficient and necessary conditions) for (a) result-preserving queries, *i.e.,* queries covered by available BaaV stores, (b) scan-free queries, *i.e.,* queries that can be evaluated without scanning any table, and (c) bounded queries, *i.e.,* queries that can be answered by accessing a bounded amount of data. We show that in parallel processing, Zidian guarantees (a) no scans for scan-free queries, (b) bounded communication cost for bounded queries; and (c) parallel scalability, *i.e.,* speed up when adding processors. Moreover, Zidian can be plugged into existing SQL-over-NoSQL systems and retains horizontal scalability. Using benchmark and real-life data, we empirically verify that Zidian improves existing SQL-over-NoSQL systems by 2 orders of magnitude on average.

PVLDB Reference Format:
Yang Cao, Wenfei Fan, Tengfei Yuan. Block as a Value for SQL over NoSQL. *PVLDB*, 12(10): 1153-1166, 2019.
DOI: https://doi.org/10.14778/3339490.3339498

1. INTRODUCTION

Key-value (KV) stores have found prevalent use in industry [22, 34, 6, 18, 7, 40]. KV stores support dictionary-like data access to retrieve and store data as key-value pairs, offering horizontal scalability, fault tolerance and transparent sharding.

To support queries at scale, several SQL engines have been developed on top of KV stores. After all, 75% of business data is generated and stored as relations [43], and analytics of the data is typically carried out via SQL queries. These systems are often based on an SQL-over-NoSQL architecture [37], which stores data persistently in a KV-store cluster, and answers queries in a computing cluster (as a separate layer) in parallel [29]. This architecture has been adopted by Google's Spanner [20, 12], Facebook's MyRocks [25], Hive [8] and SparkSQL [11], among other systems.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 12, No. 10
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3339490.3339498

While these systems offer the benefits of underlying KV-storage, they do not perform as well as traditional DBMS when evaluating SQL queries, for the following reasons.

(1) Costly scan. Typically, most SQL-over-NoSQL systems are based on a *tuple-as-a-value* (TaaV) model. It stores a relation as a set of KV pairs (k, t), in which k is an internal id or primary key of a tuple t. These KV pairs are organized in a distributed hash table (DHT). DHT supports efficient point access via `get` that given a key k, fetches the entire tuple t. However, for most SQL queries, we do not know the keys of relevant tuples in advance. Hence we have to "blindly" scan a table by incurring as many `get`'s as the size of the table.

(2) Heavy communication load. As observed by [37], few SQL-over-NoSQL systems are able to reduce data retrieval by *e.g.,* pushing selection predicates down to the storage layer, and none can execute scans efficiently. As a result, a large amount of data (even the entire relation) is often retrieved from the KV storage and is processed by the computing layer. This incurs heavy communication cost for data shuffling in parallel execution. The situation is even worse in the common practice of denormalizing databases [32, 36], *i.e.,* when using wide tables or universal relations.

Can we reduce excessive data access and communication costs, and make existing SQL-over-NoSQL systems as efficient as DBMS when it comes to answering SQL queries?

Zidian. To overcome the limitations of SQL-over-NoSQL, we develop Zidian, a middleware for KV storage. Underlying Zidian is a *block-as-a-value* model (BaaV). In contrast to the conventional TaaV model for KV stores, BaaV represents relations in KV stores as *keyed blocks* (k, B), where k is a key of a block B of partial tuples. Under BaaV, arbitrary attributes can be taken as a key k, while k can only be an id or primary-key attributes under TaaV.

Under the BaaV mode, Zidian offers the following.

[1] Efficient SQL. Zidian speeds up SQL-over-NoSQL systems by reducing `get` invocations, retrieval of irrelevant data, and hence both computation cost and communication cost.

(a) Keyed blocks offer a data locality of relations in DHT. With a single `get`, one can retrieve a block of relevant data.

(b) BaaV provides KV stores with convenient indexing, which, as observed by [37], is not yet well supported by KV stores.

By making explicit use of indexes, we can make queries *scan-free*, to be answered without scanning any table. A scan-free query Q fetches and operates on only the part of data needed for answering Q, and hence also reduces computational cost.

(c) By reasoning about keyed blocks (k, B) and the size of B, we can check whether a query is *bounded*, which only need to access a bounded amount of data no matter how big the underlying dataset is, and hence can be answered with bounded computation and communication costs.

[2] Scalability. In parallel processing, Zidian guarantees (a) *parallel scalability*, *i.e.,* speedup guaranteed when adding computing nodes to the computing layer; and (b) bounded communication cost for bounded queries. Moreover, (c) Zidian retains the *horizontal scalability* of SQL-over-NoSQL systems, *i.e.,* increasing throughput when adding new nodes to the storage layer, where the throughput is the total number of tuples retrieved from all storage nodes per second via get [37].

[3] Ease of use. Zidian can be built on top of any SQL-over-NoSQL systems over any KV stores, without the need to hack into the systems or change their underlying KV storage. That is, Zidian can be "plugged into" exiting SQL-over-NoSQL systems and help speed up their SQL query answering.

Contributions & organization. This paper proposes Zidian and justifies BaaV, from foundations to practice.

(1) Data model (Section 4). We introduce BaaV, a model to represent relations in KV stores as keyed blocks. We extend relational algebra to BaaV stores, to leverage the BaaV model when answering SQL queries. Moreover, we define scan-free queries and bounded queries in terms of BaaV query plans, to speed up SQL evaluation on SQL-over-NoSQL systems.

(2) A framework (Section 5). Based on BaaV, we propose Zidian, a framework to speed up SQL evaluation of existing SQL-over-NoSQL systems. It maps conventional databases $\mathcal{D}$ to BaaV stores $\vec{\mathcal{D}}$. It takes SQL queries Q posed on $\mathcal{D}$ and answers Q in the corresponding BaaV store $\vec{\mathcal{D}}$ whenever possible. We study fundamental problems underlying the framework. In particular, we provide a characterization of preservation, *i.e.,* a sufficient and necessary condition, for deciding whether a query Q posed on $\mathcal{D}$ can be answered in available $\vec{\mathcal{D}}$.

(3) Scan-free data access (Section 6). We characterize scan-free (resp. bounded) queries, *i.e.,* we develop a sufficient and necessary condition for deciding whether an SQL query is scan-free (resp. bounded) on a BaaV store. While the problem is undecidable, the characterization provides an effective syntax of such queries that can be efficiently checked. Moreover, we provide an algorithm to generate query plans, which guarantee to avoid scan (resp. access a bounded amount of data) for scan-free (resp. bounded) queries.

(4) Parallelization: boundedness and scalability (Section 7). We propose to interleave data access and computation when answering a query in parallel, instead of first fetching all the data and then computing answers. With this strategy, we show that Zidian needs no scan for scan-free queries and incurs bounded communication cost for bounded queries. Moreover, under BaaV, Zidian guarantees parallel scalability and retains the horizontal scalability of SQL-over-NoSQL systems.

(5) Implementation (Section 8). As a proof of concept, we have implemented Zidian and deployed it for SoH (SparkSQL-over-HBase [7]), SoK (SparkSQL-over-Kudu [7]) and SoC (SparkSQL-over-Cassandra [6]). In addition to the framework of Section 5, Zidian also includes (a) a module to help design BaaV schema under storage constraints (Section 8.1); and (b) adapters for deploying Zidian over existing KV systems.

(6) Experiments (Section 9). Using benchmark TPC-H [42] and real-life data, we evaluate the effectiveness of Zidian. We find the following on average. (1) Zidian outperforms SoH, SoK and SoC in efficiency by $2.8{\times}10^2$, $1.7{\times}10^2$ and $8.1{\times}10^2$ times for scan-free queries, respectively, and by $2.0{\times}10^2$, $1.5{\times}10^2$ and $3.6{\times}10^2$ times for non scan-free queries. (2) With Zidian the systems incur stable computation and communication costs for bounded queries when datasets grow. (3) Zidian is parallel scalable and scales well with datasets, *e.g.,* on average Zidian on top of SoH takes 27.7 and 65.4 seconds for scan-free and non scan-free queries on datasets of 128GB with 8 workers, respectively, compared to $1.7{\times}10^3$ and $2.1{\times}10^3$ seconds by SoH without Zidian. (4) Zidian retains the horizontal scalability of underlying KV systems for KV workload.

We discuss related work in Section 2 and review SQL-over-NoSQL in Section 3. The proofs of the results are in [2].

2. RELATED WORK

We categorize related work as follows.

SQL-over-NoSQL. The SQL-over-NoSQL architecture is widely used to support scalable parallel SQL processing over commodity machines, *e.g.,* [34, 12, 40, 19, 35, 41, 25], capitalizing on KV systems as the storage, such as Apache's Cassandra [6], HBase [7] and Kudu [1]. Spanner [20, 12, 40] started this line of work, to support distributed transactions at scale. It is based on BigTable [18], which stores relations as tables of KV pairs under TaaV. The work was followed by open-source systems CockroachDB [19], Nuodb [35], MyRocks [25], and Partiqle [41] (supporting SPJ). SparkSQL [11] and Hive [8] also provide SQL-like query interface for Spark and Hadoop-based KV systems over KV datasets. All these systems follow the column-family design of BigTable [18], by treating each tuple as a value with a designated row key.

While these SQL-over-NoSQL systems are able to scale with OLTP transactions, their efficiency suffers from scans on KV stores, as observed by recent attempts to support analytical (OLAP) queries [37, 15, 33]. To overcome the limitations, [37, 15, 33, 1] improve the performance of scans by designing new KV systems. They focus on exploring the design space of KV systems, to trade scan efficiency with other system parameters, *e.g.,* updating, versioning and query types. Among them, Tell [37] (a recent modern KV system optimized for scans) and Apache's Kudu [1] also explore columnar based storage for relations in KV-stores. The efforts do not help existing KV stores and SQL-over-NoSQL combinations.

This work takes a different approach, by proposing the BaaV model. It aims to improve the performance of analytical queries on existing SQL-over-NoSQL systems, without hampering their scalability. It explores new logical representation model of relations in KV stores that can be readily supported by existing systems, and studies its impact on query evaluation, without changing the KV storage.

Secondary index. The BaaV model provides the capacity of secondary index for KV stores, but it is way beyond just indexing. Few KV stores support indices. Among the few that do, secondary indexes are encoded as relations sorted by padded keys [38], and hence are still subject to the restrictions

of the TaaV model. More specifically, a secondary index on non-key attribute A of a relation R in KV stores is typically implemented as a collection of KV pairs (k, v), where keys k are A-values padded with an internal id attribute I (or a primary key of R), so that AI values are distinct under TaaV, and hence can be used as keys; they fetch entire tuples of R. This is inefficient since (a) point access on A still incurs many get invocations, (b) it does not help scans, and (c) it introduces extra index maintenance cost.

In contrast, (a) BaaV supports indexing by using DHT of KV systems, and needs only one get to fetch a block of values for the same point access on attribute A. Moreover, it reduces duplicated and unnecessary attributes in tuples fetched, and thus reduces data access and intermediate relations. The redundancies get inflated rapidly with joins. (b) It improves scans by increasing the data locality and throughput of get, while retaining the benefits of horizontal and parallel scalability. (c) Above all, as a data model BaaV exposes such indexes as "schemas", and allows users to make explicit use of the indexes for optimization. (d) Better yet, we can deduce scan-free queries and bounded queries, all at the query level. These substantially improve both computation and communication. These are beyond the scope of traditional secondary indexing that aims to speed up data retrieving only.

Materialized views. Materialized views are used in DBMS to tailor database storage and speed up query evaluation [39]. In some sense, BaaV and Zidian offer the functionality of "materialized views" for KV-stores. However, there are key differences. (a) To the best of our knowledge, no major SQL-over-NoSQL systems support and use materialized views over NoSQL storage yet. (b) One might want to extend existing KV systems to support materialized views as DBMS does. However, such an extension does not provide the benefits of BaaV-stores if the views are stored under the TaaV model. Indeed, views are essentially relations tailored for given queries in DBMS. Hence, views over KV storage (if supported) are also subject to the same limitations that base relations suffer in KV-stores under the conventional TaaV model. Therefore, BaaV is an alternative and more efficient way to support materialized views (and base relations) in KV-stores.

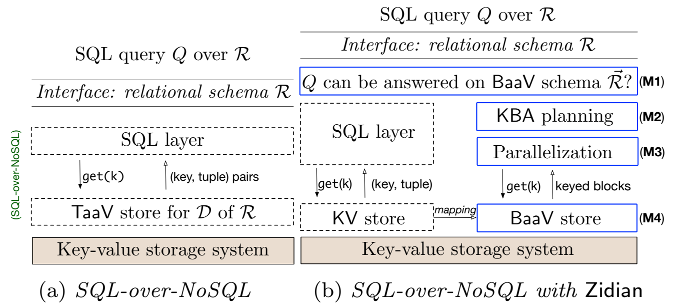
Bounded evaluation. Related to this work is also the study of bounded evaluation [26], to formalize scale independence under cardinality constraints [9, 10, 27, 16]. That line of work adopts a hash-based index guided by the cardinality constraints to determine whether only a bounded amount of data is required for answering relational queries with index-only plans, by query rewriting under cardinality constraints.

This work differs from bounded evaluation in the following. (a) The focus of bounded evaluation is to decide what query can be boundedly evaluated given a set of cardinality constraints and their associated hash-based indices. In contrast, we do not require the availability of cardinality constraints. (b) Bounded evaluation works on DBMS only, while BaaV and Zidian are developed for KV stores in SQL-over-NoSQL. (c) Bounded evaluation did not study, *e.g.,* algebra, parallelization and data mapping, which we develop for Zidian.

3. PRELIMINARIES

We review basic notations for SQL-over-NoSQL systems.

key-value storage. A KV store is a collection of key-value (KV) pairs (k, v), referred to as *key* and *value attributes*, respectively. It supports (a) $\texttt{get}(k)$ to retrieve a KV pair (k, v) with $\texttt{key} = k$, (b) $\texttt{put}(k, v)$ to add a KV pair, and (c) $\texttt{next}()$ to iterate over all keys and get the next key.



(a) *SQL-over-NoSQL* (b) *SQL-over-NoSQL with* Zidian

Figure 1: Improving SQL-over-NoSQL using Zidian

Relations in KV stores. A tuple t of a relation R is represented in KV stores under the TaaV (tuple-as-a-value) model as a KV pair (k, v), where k is an id or the primary key of R in t, and v is t. Relation R is stored as a set of KV pairs sharing the same key and value attributes, in which each pair represents a tuple of R. KV stores of relations are typically encapsulated and referred to as, *e.g.,* wide-column family stores since they provide tableau views of relations.

A *scan* of R is carried out by invoking get operations with keys extracted via $\texttt{next}()$, iterating over all keys in R.

SQL-over-NoSQL. In such a system, as shown in Fig. 1a, a database $\mathcal{D}$ of relational schema $\mathcal{R}$ is stored as KV stores in the storage layer, under the TaaV model. An SQL-over-NoSQL system exposes $\mathcal{R}$ to the users, who can then issue SQL queries Q over $\mathcal{R}$. The evaluation of Q is conducted in a separated layer called *the SQL layer*, by a computing cluster. The SQL layer consists of SQL parser, planner and executor, to generate a query plan ξ for Q. The plan ξ accesses data in the underlying KV store of $\mathcal{D}$ via get operations only.

An SQL-over-NoSQL system works as follows. Upon receiving an SQL query Q, the storage layer retrieves all relations involved in Q and moves the data to the SQL layer; the SQL layer then generates a parallel query plan ξ for Q, and executes the plan on all computing nodes in parallel.

The separation of storage and computation gives SQL-over-NoSQL (a) high availability since heavy computation tasks do not affect storage, and (b) easy scalability since we can scale out and in on demand. However, it comes with a price: data access typically incurs slow full-relation scans and the communication load for the SQL layer is hence heavy.

4. THE BAAV MODEL

In this section, we first introduce the BaaV model, to represent relations in KV stores as keyed blocks (Section 4.1). We then propose an algebra over BaaV stores (Section 4.2).

4.1 BaaV: Block-as-a-Value

BaaV stores. A KV *schema* is of the form $\vec{R}\langle X, Y\rangle$, where X and Y are sets of attributes. A *keyed block* over $\vec{R}\langle X, Y\rangle$ is a KV pair (k, B), where k is a tuple over attributes X and B is a *set* of tuples over attributes Y. A KV *instance* $\vec{D}$ of $\vec{R}\langle X, Y\rangle$ is a set of keyed blocks over $\vec{R}$ with *distinct* keys.

The *degree* of $\vec{D}$, denoted by $\texttt{deg}(D)$, is measured as the maximum size of keyed blocks (k, B) in $\vec{D}$, *i.e.,* $\texttt{deg}(D) = \max_{(k,B)\in\vec{D}} |B|$, where $|B|$ is the number of tuples in B.

Table 1: A summary of notations

Notation	Definition
$R(Z)$	relation schema under TaaV
$\vec{R}\langle X, Y\rangle$	KV schema under BaaV
$\mathcal{R}$ (resp. $\vec{\mathcal{R}}$)	relational schema (BaaV schema)
$\mathsf{att}(\vec{R})$ (resp. $\mathsf{pk}(\vec{R})$)	attributes (resp. primary key) of $\vec{R}$
$\mathcal{D}$ (resp. D)	relational database (resp. relations)
$\lvert D\rvert$ (resp. $\lVert D\rVert$)	number of tuples (resp. values) in D
$\vec{\mathcal{D}}$ (resp. $\vec{D}$)	a BaaV store (resp. KV) instance

Example 1: Consider $\mathcal{R}_1$ with TPC-H relations (simplified) SUPPLIER(suppkey, nationkey), PARTSUPP(partkey, suppkey, supplycost, availqty), and NATION(nationkey, name), in which primary keys are underlined. Under the BaaV model, they can be stored in KV storage with the following KV schemas:

$\vec{\text{SUPPLIER}}\langle$nationkey, suppkey$\rangle$,
$\vec{\text{PARTSUPP}}\langle$suppkey, (partkey, supplycost, availqty)$\rangle$, and
$\vec{\text{NATION}}\langle$name, nationkey$\rangle$.

Note that under the TaaV model, nationkey, suppkey, and name cannot be key attributes since they are not the primary keys of corresponding relations. In contrast, under BaaV, they are taken as keys since BaaV values are blocks of tuples. □

A KV schema $\vec{R}\langle X, Y\rangle$ can also carry a primary key, which is a subset $W \subseteq XY$ of attributes such that for any of its KV instance $\vec{D}$ and any Y-tuples t_1 and t_2 associated with the same key (*i.e.,* X-tuple), t_1 and t_2 are distinct on $W \cap Y$ attributes. Note that it is not necessary that $W \subseteq X$.

A BaaV *schema* $\vec{\mathcal{R}}$ is a set of KV schemas. A BaaV *store* $\vec{\mathcal{D}}$ of $\vec{\mathcal{R}}$ consists of all KV instances of KV schemas in $\vec{\mathcal{R}}$.

The degree of $\vec{\mathcal{D}}$, denoted by $\mathsf{deg}(\vec{\mathcal{D}})$, is the maximum degree of KV instances in $\vec{\mathcal{D}}$, *i.e.,* $\mathsf{deg}(\vec{\mathcal{D}}) = \max_{\vec{D}\in\vec{\mathcal{D}}} \mathsf{deg}(\vec{D})$.

Properties. BaaV offers the following benefits.

(1) In contrast to TaaV, BaaV stores allow arbitrary attributes to be taken as keys. In fact, TaaV is a special case of BaaV when in a keyed block (k, B), B is a single tuple.

(2) Each `get` invocation can retrieve more data under BaaV than under TaaV, and is hence more efficient on BaaV stores.

(3) The degree of BaaV stores indicates the extent of data locality of the DHT of KV storage. By tuning the degree of BaaV stores, we can get bounded queries, and balance the efficiency and update cost of KV stores under BaaV.

Mapping between relational databases and KV stores. There is a convenient correspondence between the two.

(1) Consider an instance $\vec{D}$ of KV schema $\langle X, Y\rangle$, *i.e.,* a set of key-block pairs (k, B). For each Y-tuple t in block B, we refer to (k, t) as *a tuple of* $\vec{D}$. The *relational version* D of $\vec{D}$ is the set of all tuples of $\vec{D}$. It is an instance of traditional relation schema (X, Y), by flattening B. Similarly we define the *relational version* $\mathcal{D}$ of a BaaV store $\vec{\mathcal{D}}$.

(2) Consider a relational schema $\mathcal{R}$, a database $\mathcal{D}$ of $\mathcal{R}$, and a BaaV schema $\vec{\mathcal{R}}$. The *mapping* of $\mathcal{D}$ on $\vec{\mathcal{R}}$ is a BaaV store $\vec{\mathcal{D}}$ of $\vec{\mathcal{R}}$ defined as follows: for each instance D of $R \in \mathcal{R}$ in $\mathcal{D}$ and any KV schema $\vec{R}\langle X, Y\rangle \in \vec{\mathcal{R}}$, if XY are attributes in R, then $\vec{\mathcal{D}}$ includes a KV instance $\vec{D}$ obtained from D by first projecting D on XY and then grouping by X. For convenience, in the sequel we assume that each $\vec{R}$ in $\vec{\mathcal{R}}$ is composed of attributes from the same relation schema of $\mathcal{R}$.

Notations are summarized in Table 1.

A	B
1	1 2
2	3 4

$\vec{R}_1\langle A \lhd B\rangle$

B	C
1	1 2
3	3 4

$\vec{R}_2\langle B \lhd C\rangle$

A	C
1	1 4
2	2 3

$\vec{R}_3\langle A \lhd C\rangle$

A B	C
1 1	1 2
2 3	3 4

$\vec{R}_4 = \vec{R}_1 \propto \vec{R}_2$

A	B C
1	1 1 1 2
2	3 3 3 4

$\vec{R}_5 = \vec{R}_4 \uparrow_A$

Figure 2: KV instances in Example 2

4.2 KBA: An Algebra of Keyed Blocks

As opposed to relations, queries on BaaV stores are evaluated on keyed blocks. In light of this, we present KBA, an extension of relational algebra (RA) to express internal query plans on BaaV stores. Below we first present KBA algebra, and then study properties of KBA plans over BaaV stores.

KBA **algebra**. We present major operators of KBA.

(1) Extension ($\propto$). Consider instances $\vec{D}_1$ of KV schema $\langle X, Y\rangle$ and $\vec{D}_2$ of $\langle Y', Z\rangle$. If $Y' \subseteq XY$, then the *extension* of $\vec{D}_1$ with $\vec{D}_2$, denoted by $\vec{D}_1 \propto \vec{D}_2$, is the mapping of $D_1 \bowtie_{Y'} D_2$ on a new KV schema $\langle XY, Z\rangle$. Here $D_i (i = \{1, 2\})$ is the relational version of KV instance $\vec{D}_i$, and $\bowtie_{Y'}$ is the natural join in the classical RA on attributes Y'.

Intuitively, $\vec{D}_1 \propto \vec{D}_2$ extends $\vec{D}_1$ with relevant Z-attributes of $\vec{D}_2$, extracted by using values of $\vec{D}_1$ as keys. One can view $\vec{D}_1 \propto \vec{D}_2$ as a special "join" unique to BaaV: it joins $\vec{D}_1$ and $\vec{D}_2$ but it does not access all the data in $\vec{D}_2$, *i.e.,* it does not scan $\vec{D}_2$. We will show that $\propto$ allows us to express query plans whose execution does not incur blind scans over BaaV stores.

(2) Shift ($\uparrow$). For KV instance $\vec{D}$ of $\langle X, Y\rangle$ and $X' \subseteq XY$, the *shift* of $\vec{D}$ with X', denoted by $\vec{D}\uparrow_{X'}$, is a KV instance $\vec{D}'$ of $\langle X', XY \setminus X'\rangle$ that has the same relational version as $\vec{D}$.

Intuitively, $\vec{D}\uparrow_{X'}$ redistributes the key and value attributes for $\vec{D}$, by setting X' as key. It allows us to align KV instances and enables set-like operations (*e.g.,* union and difference) on KV instances with different key attribute distribution.

Extension ($\propto$) and shift ($\uparrow$) are new operators unique to KBA that find no counterpart in RA. We next introduce extensions of conventional RA operators for the BaaV model.

(3) Join ($\bowtie$). For KV instances $\vec{D}_1$ of $\langle X_1, Y_1\rangle$ and $\vec{D}_2$ of $\langle X_2, Y_2\rangle$, the *join* of $\vec{D}_1$ and $\vec{D}_2$ on attributes X ($X \subseteq X_1Y_1 \cap X_2Y_2$), denoted by $\vec{D}_1 \bowtie_X \vec{D}_2$, is the mapping of the join of relational versions of $\vec{D}_1$ and $\vec{D}_2$ on KV schema $\langle X_1X_2, Y_1Y_2\rangle$.

Example 2: Consider KV instances of $\vec{R}_1\langle A, B\rangle$, $\vec{R}_2\langle B, C\rangle$, $\vec{R}_3\langle A, C\rangle$ shown in Fig. 2 (left side). The results of $\vec{R}_1 \propto \vec{R}_2$ and $\vec{R}_4 \uparrow_A$ are the instances of $\vec{R}_4\langle AB, C\rangle$ and $\vec{R}_5\langle A, BC\rangle$, respectively, in Fig. 2 (right side). The join result of $\vec{R}_5 \bowtie_{AC} \vec{R}_3$ contains keyed blocks (1, {(1, 1)}) and (2, {(3, 3)}). □

Other relational operators are defined along the same lines. By transforming between KV instances and relations on the fly (see Section 4.1), KBA queries can readily benefit from optimizations developed for RA. KBA is closed and is relational-complete. Indeed, it can express $\mathsf{RA}_{\mathsf{aggr}}$, *i.e.,* the extension of RA with group-by aggregates: for any database $\mathcal{D}$, any BaaV-store $\vec{\mathcal{D}}$ such that $\mathcal{D}$ is a relational version of $\vec{\mathcal{D}}$, for any $\mathsf{RA}_{\mathsf{aggr}}$ Q over $\mathcal{D}$, there exists a KBA $\vec{Q}$ such that $Q(\mathcal{D}) = \vec{Q}(\vec{\mathcal{D}})$, *i.e.,* answer $Q(\mathcal{D})$ is the relational version of answer $\vec{Q}(\vec{\mathcal{D}})$.

KBA **plans**. A KBA plan is similar to an RA plan tree [3]. As opposed to a conventional RA plan whose leaf nodes are relations, in a KBA plan, (1) each leaf is either a constant

(constant keyed block) or a KV instance; and (2) each intermediate node is one of the operators of KBA. In particular, for KBA plan $\xi = \vec{R}_1 \propto \vec{R}_2$, $\vec{R}_1$ is the unique leaf node of ξ, and $\vec{R}_2$ is treated as a parameter of $\propto$.

Denote by $\xi(\vec{\mathcal{D}})$ the execution result of ξ over BaaV store $\vec{\mathcal{D}}$, where ξ and $\vec{\mathcal{D}}$ are over the same BaaV schema $\vec{\mathcal{R}}$.

Below we identify two special classes of KBA plans.

Scan-free KBA *plans*. A KBA plan ξ is *scan-free* over BaaV schema $\vec{\mathcal{R}}$ if all its leaf nodes are constants. Intuitively, the execution of such ξ incurs *no scans* on any BaaV store of $\vec{\mathcal{R}}$.

Example 3: Recall $\mathcal{R}_1$ and the KV schemas $\vec{\text{PARTSUPP}}$, $\vec{\text{SUPPLIER}}$ and $\vec{\text{NATION}}$ from Example 1. Let $\vec{\mathcal{R}}_1$ be a BaaV schema consisting of these KV schemas. Consider SQL Q_1 over $\mathcal{R}_1$, which is a simplified TPC-H query (query q_{11} [42]):

```
select     PS.suppkey, SUM(PS.supplycost)
from       PARTSUPP as PS, SUPPLIER as S, NATION as N
where      PS.suppkey = S.suppkey and S.nationkey
           = N.nationkey and N.name = "GERMANY"
group by   PS.suppkey
```

Consider KBA plans $\xi_1' = ((\text{"Germany"} \propto \vec{\text{NATION}}) \propto \vec{\text{SUPPLIER}}) \propto \vec{\text{PARTSUPP}}$ and $\xi_1 = \text{group_by}(\xi_1', \text{PS.suppkey}, \text{SUM}(\text{PS.supplycost}))$ (group ξ_1' by PS.suppkey and sum over PS.supplycost per group). Then ξ_1 is scan-free over $\vec{\mathcal{R}}_1$ since it uses $\propto$ only to access all KV instances. Moreover, on corresponding BaaV stores ξ_1 answers Q_1: for any database $\mathcal{D}$ of $\mathcal{R}_1$, $\xi_1(\vec{\mathcal{D}}) = Q_1(\mathcal{D})$, where $\vec{\mathcal{D}}$ is the mapping of $\mathcal{D}$ on $\vec{\mathcal{R}}_1$. ☐

Bounded KBA *plans*. A KBA plan ξ is *bounded* over BaaV store $\vec{\mathcal{D}}$ of schema $\vec{\mathcal{R}}$ if (a) ξ is scan-free over $\vec{\mathcal{R}}$ and (b) $\vec{\mathcal{D}}$ has a degree bounded by a constant independent of $|\vec{\mathcal{D}}|$.

Intuitively, if ξ is bounded over $\vec{\mathcal{D}}$, then ξ on $\vec{\mathcal{D}}$ is guaranteed to access a bounded amount of data.

Scan-free and bounded SQL queries. As will be seen shortly, Zidian aims to take as input SQL queries Q posed on a database $\mathcal{D}$ of relational schema $\mathcal{R}$, and answer Q over $\vec{\mathcal{D}}$ of BaaV schema $\vec{\mathcal{R}}$, where $\vec{\mathcal{D}}$ is the mapping of $\mathcal{D}$ on $\vec{\mathcal{R}}$.

We say that an SQL query Q over $\mathcal{R}$ *can be answered over* $\vec{\mathcal{R}}$ if it has a KBA plan ξ over $\vec{\mathcal{R}}$ *i.e.,* for any instance $\mathcal{D}$ of $\mathcal{R}$, $Q(\mathcal{D}) = \xi(\vec{\mathcal{D}})$. An SQL query Q over $\mathcal{R}$ is *scan-free* over BaaV schema $\vec{\mathcal{R}}$ if it has a scan-free KBA plan over $\vec{\mathcal{R}}$. Similarly, we define *bounded* SQL queries over BaaV store $\vec{\mathcal{D}}$.

5. SQL-OVER-NOSQL WITH BAAV

In this section we show how to improve existing SQL-over-NoSQL systems using BaaV. Below we first propose Zidian, a middleware framework that extends the SQL-over-NoSQL architecture with BaaV (Section 5.1). We then study some fundamental problems underlying Zidian (Sections 5.2).

5.1 Zidian: SQL-over-NoSQL with BaaV

As shown in Fig. 1b, Zidian is built on top of the KV storage of the SQL-over-NoSQL architecture, and extends it with support of the BaaV model. By deploying Zidian over existing SQL-over-NoSQL systems, BaaV speeds up SQL query answering, without modifying the KV storage.

More specifically, Zidian extends SQL-over-NoSQL with four major modules **M1**–**M4** shown in Fig. 1b, as follows.

(1) At the bottom (**M4**), Zidian maintains a BaaV store $\vec{\mathcal{D}}$, which is mapped from a conventional database $\mathcal{D}$ of schema $\mathcal{R}$ to a BaaV schema $\vec{\mathcal{R}}$. BaaV store $\vec{\mathcal{D}}$ is physically stored in the same KV system. Zidian helps users design $\vec{\mathcal{R}}$ and construct $\vec{\mathcal{D}}$, based on an analysis of historical query patterns. Users may opt to build $\vec{\mathcal{D}}$ to cover all the attributes of $\mathcal{D}$, and then drop $\mathcal{D}$ entirely. Alternatively, they may pick $\vec{\mathcal{D}}$ to cover selected attributes only, and keep $\vec{\mathcal{D}}$ small. When $\vec{\mathcal{D}}$ is in place, Zidian incrementally maintains $\vec{\mathcal{D}}$ in response to updates.

(2) At the top (**M1**), Zidian allows users to pose conventional SQL queries Q on relational database $\mathcal{D}$. It checks whether Q can be answered in BaaV store $\vec{\mathcal{D}}$. If so, it passes Q to **M2** (see below); otherwise, Zidian identifies sub-queries of Q that can be answered on $\vec{\mathcal{D}}$, passes them to **M2**, and processes the rest of Q using the existing SQL layer directly.

(3) Zidian extends existing SQL layer to generate KBA plans ξ for queries Q that can be answered in available BaaV store $\vec{\mathcal{D}}$ (**M2**). It identifies Q that is *scan-free* over $\vec{\mathcal{R}}$ or *bounded* over $\vec{\mathcal{D}}$, and guarantees to generate scan-free and bounded KBA plans for such queries, respectively. For queries Q that are not scan-free but can be answered over $\vec{\mathcal{R}}$, Zidian identifies sub-queries of Q that are scan-free and generates KBA plans for Q with scan-free sub-plans for such sub-queries.

(4) Upon receiving a KBA plan ξ, Zidian generates a parallel KBA plan ξ_p by parallelizing ξ (**M3**). It guarantees the following: (a) parallel scalability, (b) if ξ is scan-free over $\vec{\mathcal{R}}$, then ξ_p scans no table, and (c) if ξ is bounded over $\vec{\mathcal{D}}$, then ξ_p incurs bounded communication cost. It also retains the horizontal scalability of existing SQL-over-NoSQL systems.

5.2 Fundamental Problems

We next study fundamental problems underlying module **M1** of Zidian. We will study problems associated with **M2**, **M3** and **M4** in Sections 6, 7 and 8, respectively.

Given an SQL query Q posed on a database D of schema $\mathcal{R}$, Zidian checks whether Q can be answered in the BaaV store $\vec{\mathcal{D}}$ of schema $\vec{\mathcal{R}}$ (module **M1**). The need for this is evident when $\vec{\mathcal{R}}$ covers only those attributes selected by users.

We say that $\vec{\mathcal{R}}$ is *result preserving* for query Q over $\mathcal{R}$ if there exists a KBA plan ξ over $\vec{\mathcal{R}}$ such that for any database $\mathcal{D}$ of $\mathcal{R}$, $\xi(\vec{\mathcal{D}}) = Q(\mathcal{D})$, where $\vec{\mathcal{D}}$ is the mapping of $\mathcal{D}$ on $\vec{\mathcal{R}}$.

A special case is *data preserving*, when $\vec{\mathcal{R}}$ is result preserving for *all* SQL *queries* over $\mathcal{R}$. Intuitively, a data preserving $\vec{\mathcal{R}}$ preserves all information of $\mathcal{D}$ in its mapping $\vec{\mathcal{D}}$ on $\vec{\mathcal{R}}$.

Below we characterize the data preservability and result preservability, based on which Zidian implements **M1**.

(1) Data preservability. Condition (I) below gives a characterization, where $\mathcal{R}$ is a database schema and $\vec{\mathcal{R}}$ is a BaaV schema; $\text{pk}(\vec{R})$ denotes the primary key of $\vec{R}$, and $\text{att}(\vec{R})$ denotes the set of all attributes of $\vec{R}$; similarly for $\text{att}(R)$.

Condition (I): Data preservability

For each relation $R \in \mathcal{R}$, there exists a KV schema $\vec{R} \in \vec{\mathcal{R}}$ such that $\text{att}(R) = \text{clo}(\vec{R}, \vec{\mathcal{R}})$, where $\text{clo}(\vec{R}, \vec{\mathcal{R}})$ is inductively defined as:

(1) $\text{att}(\vec{R}) \subseteq \text{clo}(\vec{R}, \vec{\mathcal{R}})$; and

(2) if $\text{pk}(\vec{R'}) \subseteq \text{clo}(\vec{R}, \vec{\mathcal{R}})$ for $\vec{R'} \in \vec{\mathcal{R}}$, then $\text{att}(\vec{R'}) \subseteq \text{clo}(\vec{R}, \vec{\mathcal{R}})$.

Theorem 1: *Condition (I) is a sufficient and necessary condition for* $\vec{\mathcal{R}}$ *to be data preserving for* $\mathcal{R}$. ☐

Example 4: BaaV schema $\vec{\mathcal{R}}_1$ of Example 3 is data preserving for $\mathcal{R}_1$ by **Condition (I)**: $\text{att}(\vec{\text{NATION}}) = \text{clo}(\text{NATION}, \vec{\mathcal{R}}_1) = \text{att}(\text{NATION})$; similarly for $\vec{\text{SUPPLIER}}$, $\vec{\text{PARTSUPP}}$. ☐

Complexity. Based on Theorem 1, Zidian checks whether BaaV schema $\vec{\mathcal{R}}$ is data preserving for $\mathcal{R}$ in $O(|\mathcal{R}||\vec{\mathcal{R}}|^2)$-time, where $|\mathcal{R}|$ is the number of attributes in $\mathcal{R}$ (see [2] for details).

Remark. Theorem 1 shows that KBA is relationally complete: any SQL query Q on $\mathcal{D}$ can be *exactly* answered by transforming it to a KBA plan ξ on $\vec{\mathcal{D}}$ such that $Q(\mathcal{D}) = \xi(\vec{\mathcal{D}})$ if the KBA schema is data preserving. One can verify by contradiction that if any operator is dropped from KBA, it can no longer ensure the relational completeness.

(2) Result preservability. We next characterize when $\vec{\mathcal{R}}$ is result preserving for Q over $\mathcal{R}$. We first study SPC queries, and then extend to $\mathsf{RA_{aggr}}$ (RA with group-by aggregates).

The characterization uses the following notations.

(1) Denote by $\mathsf{min}(Q)$ *the minimal equivalent query* of SPC query Q. Informally, $\mathsf{min}(Q)$ is obtained from Q by removing all redundant relations (see [3] for details). For example, for $Q = \pi_A(R_1(A, B) \bowtie R_2(A, B))$ where R_1 and R_2 rename relation $R(A, B)$, either R_1 or R_2 could be removed without changing answers to Q. For an SPC query Q, there exists a unique minimal equivalent query up to isomorphism [3].

(2) Denote by X_R^Q the set of attributes of relation R that appear in selection/join predicates or the final projection of Q.

Condition (II) below characterizes the whether BaaV schema $\vec{R}$ is result preserving for SPC query Q over $\mathcal{R}$.

Condition (II): Result preservability

For each relation R in the minimal equivalent query $\mathsf{min}(Q)$ of SPC Q, there exists $\vec{R} \in \vec{\mathcal{R}}$ such that $X_R^{\mathsf{min}(Q)} \subseteq \mathsf{clo}(\vec{R}, \vec{\mathcal{R}})$.

Theorem 2: *Condition (II) is a sufficient and necessary condition for $\vec{\mathcal{R}}$ to be result preserving for* SPC *query Q.* ☐

Example 5: Continuing with Example 3, consider SPC query Q'_1 that is simply Q_1 without the final group-by operation; *i.e.,* Q'_1 is $\pi_{\mathsf{PS.suppkey,PS.supplycost}}\sigma_C(\mathsf{N}\times\mathsf{S}\times\mathsf{PS})$, where N, S and PS are short for NATION, SUPPLIER and PARTSUPP, respectively; condition C is N.name = "GERMANY" $\wedge$ N.nationkey = S.nationkey $\wedge$ S.suppkey = PS.suppkey. Consider BaaV schema $\vec{\mathcal{R}}'_1$, which is derived from $\vec{\mathcal{R}}_1$ by replacing $\vec{\mathsf{PARTSUPP}}$ with $\vec{\mathsf{PARTSUPP}}'\langle\mathsf{suppkey}, (\mathsf{partkey}, \mathsf{supplycost})\rangle$.

Unlike $\vec{\mathcal{R}}_1$, $\vec{\mathcal{R}}'_1$ is not data preserving for $\mathcal{R}_1$ by Condition (I). However, $\vec{\mathcal{R}}'_1$ is result preserving for Q'_1 by Condition (II). Indeed, Q'_1 is a minimal SPC and $X_{\mathsf{PS}}^{Q'_1} = \mathsf{clo}(\vec{\mathsf{PARTSUPP}}', \vec{\mathcal{R}}'_1) = \{\mathsf{suppkey}, \mathsf{supplycost}\}$; similarly for $X_{\mathsf{S}}^{Q'_1}$ and $X_{\mathsf{N}}^{Q'_1}$.

Consider SPC $Q_2 = \pi_{\mathsf{PS.suppkey,PS.supplycost}}\sigma_{C'}(\mathsf{N} \times \mathsf{S} \times \mathsf{PS} \times \mathsf{PS}')$, where PS′ renames PARTSUPP and $C' = C \wedge$ PS.availqty = PS′.availqty. Then $X_{\mathsf{PS}}^{Q_2} = \{\mathsf{suppkey}, \mathsf{supplycost}, \mathsf{availqty}\} \not\subseteq \mathsf{clo}(\vec{\mathsf{PARTSUPP}}', \vec{\mathcal{R}}'_1)$. However, $\mathsf{min}(Q_2) = Q'_1$. Hence $X_{\mathsf{PS}}^{\mathsf{min}(Q_2)} = \mathsf{clo}(\vec{\mathsf{PARTSUPP}}', \vec{\mathcal{R}}'_1)$ and similarly for $X_{\mathsf{N}}^{\mathsf{min}(Q_2)}$ and $X_{\mathsf{S}}^{\mathsf{min}(Q_2)}$. Thus, $\vec{\mathcal{R}}'_1$ is also result preserving for Q_2. This justifies query minimization in Condition (II). ☐

Complexity. It is NP-complete to decide whether $\vec{\mathcal{R}}$ is result preserving for an SPC query Q. Indeed, it is intractable to find the minimal equivalent query $\mathsf{min}(Q)$ of Q, *a.k.a.* SPC *minimization* [3]. Nonetheless, several effective algorithms for SPC minimization are already in place (see *e.g.,* [3]). Employing one of these algorithms, Zidian checks result preservability based on Condition (II) in $O(T_{\mathsf{min}}(Q) + |Q||\vec{\mathcal{R}}|^2)$-time, where $T_{\mathsf{min}}(Q)$ is the time for minimizing Q (see [2] for details).

Extending to $\mathsf{RA_{aggr}}$. We next consider $\mathsf{RA_{aggr}}$ queries, which extend RA with additionally a group-by aggregate operator:

$$\mathsf{group_by}(Q, X, \mathsf{agg}_1(V_1), \ldots, \mathsf{agg}_m(V_m)),$$

where (a) Q is an RA query, (b) X is a set of attributes for group-by, (c) agg_i is one of aggregates `max`, `min`, `count`, `SUM`, `avg`, and (d) $V_1, \ldots, V_m$ are attributes such that $X \cup \bigcup_{i=1}^m\{\mathsf{agg}_i(V_i)\}$ forms the output relation of Q [3].

When it comes to $\mathsf{RA_{aggr}}$ queries, result preservability is beyond reach: it is undecidable. Indeed, it is undecidable to check whether an $\mathsf{RA_{aggr}}$ query is satisfiable (cf. [3]), which can be readily reduced to the result preservability problem.

Not all is lost. Condition (II) provides us with an *effective syntax* of $\mathsf{RA_{aggr}}$ queries for which $\vec{\mathcal{R}}$ is result preserving. It uses *max* SPC *sub-queries.* A max SPC sub-query Q_s of an $\mathsf{RA_{aggr}}$ Q is a sub-query of Q such that Q_s is an SPC and no other SPC sub-query Q'_s of Q includes Q_s as a sub-query.

Theorem 3: BaaV *schema $\vec{\mathcal{R}}$ is result preserving for* $\mathsf{RA_{aggr}}$ *Q if and only if Q is equivalent to an* $\mathsf{RA_{aggr}}$ *Q' such that $\vec{\mathcal{R}}$ is result preserving for every max* SPC *sub-query of Q'.* ☐

Theorem 3 assures that we can focus on $\mathsf{RA_{aggr}}$ queries whose max SPC sub-queries can be answered over $\vec{\mathcal{R}}$. This does not lose expressive power, since such $\mathsf{RA_{aggr}}$ queries can express all $\mathsf{RA_{aggr}}$ queries for which $\vec{\mathcal{R}}$ is result preserving, up to equivalence. Moreover, such $\mathsf{RA_{aggr}}$ queries can be effectively checked by employing algorithms for checking the result preservability of SPC queries outlined above. Hence Zidian handles generic $\mathsf{RA_{aggr}}$ (SQL) queries based on Theorem 3.

6. SCAN-FREE DATA ACCESS

In this section we study problems underlying module **M2** of Zidian. We first characterize SQL queries posed on database $\mathcal{D}$ that are scan-free or bounded over the corresponding BaaV store $\vec{\mathcal{D}}$ (Section 6.1). We then show how Zidian generates KBA plans, which are guaranteed scan-free (resp. bounded) for scan-free (resp. bounded) queries (Section 6.2).

6.1 Deciding Scan-Free Queries

We develop a sufficient and necessary condition for query Q over $\mathcal{R}$ to be scan-free over BaaV mapping $\vec{\mathcal{R}}$ of $\mathcal{R}$. Again we start with SPC, and then extend the study to $\mathsf{RA_{aggr}}$.

The idea is to check (a) whether attributes that are needed for answering Q can be retrieved from $\vec{\mathcal{R}}$ using scan-free plans, and (b) whether their combinations are preserved. For (a), we define a set $\mathsf{GET}(Q, \vec{\mathcal{R}})$ of all attributes in Q that can be retrieved from $\vec{\mathcal{R}}$ with scan-free plans, referred to as *retrievable attributes* over $\vec{\mathcal{R}}$. For (b), we define a set $\mathsf{VC}(Q, \vec{\mathcal{R}})$ of *sets* of attributes in $\mathsf{GET}(Q, \vec{\mathcal{R}})$ whose combinations can be verified.

Retrievable attributes $\mathsf{GET}(Q, \vec{\mathcal{R}})$. Let X_C^Q be the set of attributes A in selection conditions in Q of the form $A = c$ for a constant c. The rules below define $\mathsf{GET}(Q, \vec{\mathcal{R}})$ inductively:

(a) $X_C^Q \subseteq \mathsf{GET}(Q, \vec{\mathcal{R}})$;

(b) if $A \in \mathsf{GET}(Q, \vec{\mathcal{R}})$ and $A = B$ by the selection condition C of Q via equality transitivity, then $B \in \mathsf{GET}(Q, \vec{\mathcal{R}})$;

(c) if $X \subseteq \mathsf{GET}(Q, \vec{\mathcal{R}})$, $\vec{R}\langle X, Y\rangle \in \vec{\mathcal{R}}$, then $Y \subseteq \mathsf{GET}(Q, \vec{\mathcal{R}})$.

The set $\mathsf{GET}(Q, \vec{\mathcal{R}})$ contains nothing else.

Intuitively, $\mathsf{GET}(Q, \vec{\mathcal{R}})$ starts with constant attributes X_C^Q in Q, and recursively propagates them to other attributes via extension $\propto$ over $\vec{\mathcal{R}}$, possibly using $\uparrow$ and $\bowtie$ to form key

attributes for $\propto$. If $\vec{\mathcal{R}}$ is result preserving for Q, then for any database $\mathcal{D}$ of $\mathcal{R}$, $\mathsf{GET}(Q, \vec{\mathcal{R}})$ contains all attributes of Q such that their values in $\mathcal{D}$ needed for computing $Q(\mathcal{D})$ can be fetched from the BaaV mapping $\vec{\mathcal{D}}$ of $\mathcal{D}$ via scan-free plans.

Verifiable combinations $\mathsf{VC}(Q, \vec{\mathcal{R}})$. The set $\mathsf{GET}(Q, \vec{\mathcal{R}})$ identifies attributes of Q whose values can be fetched from $\vec{\mathcal{R}}$ using scan-free plans. To answer Q, however, not only these values but also their combinations may have to be preserved.

To do this, we define a collection $\mathsf{VC}(Q, \vec{\mathcal{R}})$ of attributes sets in $\mathsf{GET}(Q, \vec{\mathcal{R}})$ such that their value combinations in $\mathcal{D}$ of $\mathcal{R}$ can be checked using scan-free access plans over the mapping $\vec{\mathcal{D}}$ of $\mathcal{D}$. Denote by $\vec{\mathcal{R}}_Q$ those KV schemas in $\vec{\mathcal{R}}$ whose attributes are all contained in $\mathsf{GET}(Q, \vec{\mathcal{R}})$, *i.e.,*

$$\vec{\mathcal{R}}_Q = \{\vec{S} \in \vec{\mathcal{R}} \mid \mathsf{attr}(\vec{S}) \subseteq \mathsf{GET}(Q, \vec{\mathcal{R}})\}.$$

Then (recall clo() from Condition (I) of Section 5.2):

$$\mathsf{VC}(Q, \vec{\mathcal{R}}) = \{\mathsf{clo}(\vec{S}, \vec{\mathcal{R}}_Q) \mid \vec{S} \in \vec{\mathcal{R}}_Q\}.$$

We next characterize scan-free SPC queries Q in terms of $\mathsf{VC}(Q, \vec{\mathcal{R}})$. Consider query Q defined over a database schema $\mathcal{R}$, and the BaaV mapping $\vec{\mathcal{R}}$ of $\mathcal{R}$. Recall the notion of minimal equivalent queries $\mathsf{min}(Q)$ from Section 5.2.

Condition (III): scan-free evaluability

For each relation R in the minimal equivalent query $\mathsf{min}(Q)$ of Q, there exists a set $W \in \mathsf{VC}(\mathsf{min}(Q), \vec{\mathcal{R}})$ such that $X_R^{\mathsf{min}(Q)} \subseteq W$.

Theorem 4: *Condition (III) is a sufficient and necessary condition for* SPC Q *to be scan-free over* $\mathcal{R}$. □

Example 6: Recall Q'_1 and $\vec{\mathcal{R}}_1$ from Examples 5 and 3, respectively. Query Q'_1 is scan-free over $\vec{\mathcal{R}}_1$ since Condition (III) holds: $\mathsf{min}(Q'_1) = Q'_1$, $\mathsf{GET}(Q'_1, \vec{\mathcal{R}}_1) = \{\mathsf{N.name}, \mathsf{N.nationkey}, \mathsf{S.nationkey}, \mathsf{S.suppkey}, \mathsf{PS.suppkey}, \mathsf{PS.supplycost}\}$, $X_{\mathsf{N}}^{Q'_1} = \{\mathsf{N.name}, \mathsf{N.nationkey}\}$, $X_{\mathsf{PS}}^{Q'_1} = \{\mathsf{PS.supplycost}, \mathsf{PS.suppkey}\}$, $X_{\mathsf{S}}^{Q'_1} = \{\mathsf{S.nationkey}, \mathsf{S.suppkey}\}$, $\mathsf{VC}(Q'_1, \vec{\mathcal{R}}_1) = \{X_{\mathsf{N}}^{Q'_1}, X_{\mathsf{PS}}^{Q'_1}, X_{\mathsf{S}}^{Q'_1}\}$. Indeed, ξ'_1 of Example 3 is a scan-free plan for Q'_1.

Similarly, recall $\vec{\mathcal{R}}'_1$ and Q_2 from Example 5. One can verify that both Q'_1 and Q_2 are also scan-free over $\vec{\mathcal{R}}'_1$. □

<u>*Complexity*</u>. It is NP-complete to check whether an SPC Q is scan-free over $\vec{\mathcal{R}}$ (see [2] for a proof). This said, based on Condition (III), Zidian makes use of existing efficient algorithms for SPC minimization (*e.g.,* [3]) to check scan-free queries in $O(T_{\mathsf{min}}(Q) + |Q||\vec{\mathcal{R}}|(|\vec{\mathcal{R}}| + |Q|))$-time, where $T_{\mathsf{min}}(Q)$ is the cost of minimizing Q. It minimizes Q to $\mathsf{min}(Q)$, computes $\mathsf{VC}(\mathsf{min}(Q), \vec{\mathcal{R}})$ in $O(|\vec{\mathcal{R}}|(|\vec{\mathcal{R}}| + |Q|))$-time, and checks for each R in Q whether $X_R^{\mathsf{min}(Q)} \subseteq W$ for some $W \in \mathsf{VC}(\mathsf{min}(Q), \vec{\mathcal{R}})$.

Extending to $\mathsf{RA}_{\mathsf{aggr}}$. For $\mathsf{RA}_{\mathsf{aggr}}$ queries, it is not surprising that it is undecidable to check whether an $\mathsf{RA}_{\mathsf{aggr}}$ query Q over $\mathcal{R}$ is scan-free over BaaV schema $\vec{\mathcal{R}}$. It involves checking whether Q is equivalent to a scan-free query $\vec{Q}$, and the query equivalence problem is undecidable (cf. [3]).

<u>*Effective syntax*</u>. Nonetheless, below we provide an *effective syntax* for scan-free $\mathsf{RA}_{\mathsf{aggr}}$ queries. Recall the notion of SPC sub-query of max $\mathsf{RA}_{\mathsf{aggr}}$ queries from Section 5.2.

Theorem 5: *An* $\mathsf{RA}_{\mathsf{aggr}}$ *query* Q *is scan-free over* $\vec{\mathcal{R}}$ *if and only if* Q *is equivalent to an* $\mathsf{RA}_{\mathsf{aggr}}$ Q' *such that every max* SPC *sub-query of* Q' *is scan-free over* $\vec{\mathcal{R}}$. □

The class of $\mathsf{RA}_{\mathsf{aggr}}$ queries Q' described in Theorem 5 makes an effective syntax for all scan-free $\mathsf{RA}_{\mathsf{aggr}}$ queries over $\vec{\mathcal{R}}$. It covers all scan-free $\mathsf{RA}_{\mathsf{aggr}}$ queries up to equivalence, and can be effectively checked by employing the checking algorithms for scan-free SPC queries. Based on this, Zidian checks whether an $\mathsf{RA}_{\mathsf{aggr}}$ (SQL) query is scan-free. For instance, Q_1 of Example 3 is in this class *w.r.t.* $\vec{\mathcal{R}}'_1$ of Example 5.

Bounded queries. As an immediate corollary of Theorem 5, given an $\mathsf{RA}_{\mathsf{aggr}}$ query Q over a database $\mathcal{D}$, Zidian decides whether Q is bounded over the corresponding BaaV store $\vec{\mathcal{D}}$ as follows: check (a) whether Q is scan-free, and (b) whether for each relation R in each max SPC sub-query of Q, $\mathsf{deg}(\vec{D}) \leq c$, where c is a predefined bound, and $\vec{D}$ is the BaaV mapping of instance D of R in $\mathcal{D}$. If so, it concludes that Q is bounded.

6.2 Generating Scan-Free KBA Plans

We next present an algorithm for generating KBA query plans for user queries. Consider a BaaV schema $\vec{\mathcal{R}}$ and an SQL query Q over database schema $\mathcal{R}$ such that $\vec{\mathcal{R}}$ is result preserving for Q. Zidian generates a KBA plan ξ_Q for Q over $\vec{\mathcal{R}}$, such that ξ_Q is scan-free (resp. bounded) when Q is scan-free (resp. bounded) by the effective syntax above.

To generate ξ_Q for Q over $\vec{\mathcal{R}}$, Zidian adopts a chase-based approach [3] following [17, 14], outlined as follows.

(1) Zidian first generates a conventional $\mathsf{RA}_{\mathsf{aggr}}$ plan ξ_Q for Q over $\mathcal{R}$. Note that leaf nodes of ξ_Q are relations in Q.

(2) Zidian then replaces all leaf nodes of ξ_Q with KBA plans. More specifically, it generates a sequence ℓ of applications of the rules for computing $\mathsf{GET}(Q, \vec{\mathcal{R}})$ and $\mathsf{VC}(Q, \vec{\mathcal{R}})$ (Section 6.1), referred to as a *chasing sequence* for Q and $\vec{\mathcal{R}}$.

While there are possibly many chasing sequences for Q and $\vec{\mathcal{R}}$, they all converge at the same $\mathsf{GET}(Q, \vec{\mathcal{R}})$ and $\mathsf{VC}(Q, \vec{\mathcal{R}})$ (see Theorem 6 and [2]). Moreover, for each leaf node R in ξ_Q such that $X_R^Q \subseteq W$ for some $W \in \mathsf{VC}(Q, \vec{\mathcal{R}})$ (recall Condition (III)), there exists a sub-sequence ℓ_R of ℓ that "proves" this. Such a proof yields a scan-free KBA plan that fetches R for Q, by interpreting steps in ℓ_R as either $\propto$ or $\bowtie$.

(3) If Q is not scan-free over $\vec{\mathcal{R}}$, then there must exist leaf nodes R in ξ_Q such that $X_R^Q \not\subseteq W$ for all $W \in \mathsf{VC}(Q, \vec{\mathcal{R}})$. For such relations R, there must exist $\vec{R} \in \vec{\mathcal{R}}$ such that $X_R^Q \subseteq \mathsf{clo}(\vec{R}, \vec{\mathcal{R}})$ by Condition (II) since $\vec{\mathcal{R}}$ is result preserving for Q. Zidian simply replaces such R in ξ_Q with $\vec{R}$.

Example 7: Recall Q_1, $\vec{\mathcal{R}}_1$ and ξ_1 from Example 3. Zidian generates the KBA plan ξ_1 for Q_1 over $\vec{\mathcal{R}}_1$ as follows.

(a) It first generates an $\mathsf{RA}_{\mathsf{aggr}}$ plan (ξ_0 in Fig. 5 of [2]) for Q_1.

(b) It then generates a chasing sequence ℓ for Q_3 over $\vec{\mathcal{R}}_1$:

$$(\emptyset, \emptyset) \xmapsto{\text{①rule (a) of GET}} (\{\mathsf{N.name}\}, \emptyset) \xmapsto[T_1: \text{`GERMANY'} \propto \vec{\mathsf{NATION}}]{\text{② rule (c) of GET, rule (1) of clo for VC}}$$

$$(\mathsf{att(N)}, \{\mathsf{att(N)}\}) \xmapsto{\text{③rule (b) of GET}} (\mathsf{att(N)} \cup \{\mathsf{S.nationkey}\}\{\mathsf{att(N)}\})$$

$$\xmapsto[T_2: T_1 \propto \vec{\mathsf{SUPPLIER}}]{\text{④ rule (c) of GET, rule (1) of clo for VC}} (\mathsf{att(N)} \cup \mathsf{att(S)}, \{\mathsf{att(N)}, \mathsf{att(S)}\})$$

$$\xmapsto{\text{⑤rule (b) of GET}} (\mathsf{att(N)} \cup \mathsf{att(S)} \cup \{\mathsf{PS.suppkey}\}, \{\mathsf{att(N)}, \mathsf{att(S)}\})$$

$$\xmapsto[T_3: T_2 \propto \vec{\mathsf{PARTSUPP}}]{\text{⑥ rule (c) of GET, rule (1) of clo for VC}} (\mathsf{att(N)} \cup \mathsf{att(S)} \cup \mathsf{att(PS)}, \{\mathsf{att(N)}, \mathsf{att(S)}, \mathsf{att(PS)}\}),$$

where $\mathsf{att}(\mathsf{N})$ is the set consisting of all attributes in relation schema NATION; similarly for $\mathsf{att}(\mathsf{S})$ and $\mathsf{att}(\mathsf{PS})$.

Intuitively, ℓ is essentially the trace of the computation of $(\mathsf{GET}(Q_1, \vec{\mathcal{R}}_1), \mathsf{VC}(Q_1, \vec{\mathcal{R}}_1))$. Initially, both of them are $\emptyset$. At step ①, N.name is added to $\mathsf{GET}(Q_1, \vec{\mathcal{R}}_1)$ by rule (a) in the definition of GET (Section 6.1). At step ②, by rule (a) of GET, N.nationkey is added to $\mathsf{GET}(Q_1, \vec{\mathcal{R}}_1)$ by KBA plan T_1 in ℓ, yielding $\mathsf{GET}(Q_1, \vec{\mathcal{R}}_1) = \mathsf{att}(\mathsf{N})$; *i.e.,* step ② encodes T_1. In addition, by rule (1) of $\mathsf{clo}(Q_1, \vec{\mathcal{R}}_1)$ (Section 5.2) in the computation of $\mathsf{VC}(Q_1, \vec{\mathcal{R}}_1)$, set $\mathsf{att}(\mathsf{N})$ is added to $\mathsf{VC}(Q_1, \vec{\mathcal{R}}_1)$, *i.e.,* the combination of (N.name, N.nationkey)-values can be verified over $\vec{\mathcal{R}}_1$; similarly for steps ③–⑥ in ℓ.

(c) It translates ℓ into KBA plans ξ_{N}, ξ_{S} and ξ_{PS} that fetch data for N, S and PS, respectively. Specifically, ξ_{N} is the plan T_1 encoded by step ② when it finds that $X_{\mathsf{N}}^{Q_1} = \mathsf{att}(\mathsf{N})$ is included in $\mathsf{VC}(Q_1, \vec{\mathcal{R}}_1)$ (recall Condition (III)). Similarly, ξ_{S} (resp. ξ_{PS}) is plan T_2 (resp. T_3) encoded by step ④ (resp. ⑥).

(d) Finally, it replaces N, S and PS of ξ_0 with ξ_{N}, ξ_{S} and ξ_{PS}, respectively, yielding KBA plan ξ_{Q_1} for Q_1 (Fig. 5 in [2]). It optimizes ξ_{Q_1} by removing duplicated sub-plans and operations, yielding exactly ξ_1 of Example 3. Along the same lines, Zidian generates a similar KBA plan Q_1 over $\vec{\mathcal{R}}'_1$ of Example 5. □

By (2) and (3), all leaf nodes in the rewritten ξ_Q are constants or KV schemas of $\vec{\mathcal{R}}$, *i.e.,* ξ_Q becomes a KBA plan.

Theorem 6: *Plan ξ_Q (1) correctly answers query Q as long as $\vec{\mathcal{R}}$ is result preserving for Q; and (2) it is scan-free (resp. bounded) when Q is in the effective syntax for scan-free (resp. bounded) $\mathsf{RA}_{\mathsf{aggr}}$ queries over $\vec{\mathcal{R}}$.* □

Intuitively, Theorem 6 assures that when Q is in the effective syntax of Theorem 3, Zidian always finds a correct KBA plan ξ_Q for Q over $\vec{\mathcal{R}}$. Moreover, ξ_Q is scan-free (resp. bounded) if Q is scan-free (resp. bounded) by Theorem 5.

7. PARALLEL EVALUATION OVER BAAV

In this section we show how KBA plans can be executed in parallel while guaranteeing parallel scalability and horizontal scalability (module **M3** of Zidian). We first review how query plans are parallelized in existing SQL-over-NoSQL systems (Section 7.1). We then propose an interleaving strategy to parallelize KBA plans and prove the guarantees (Section 7.2).

7.1 Overview

We start with the parallel strategies of SQL-over-NoSQL systems. We then show how Zidian parallelizes KBA plans.

Parallelization in SQL-over-NoSQL systems. An SQL-over-NoSQL system typically parallelizes a sequential query plan ξ as follows. (a) It first retrieves all relations that appear in ξ from the storage layer. (b) Moving the data to the SQL layer, it then executes each operator of ξ in parallel one by one. Among the RA operators, parallelization of join is the trickiest part. The common practice is to use parallel hash join algorithms [31]. While there have been recent theoretical proposals on hypercube-based, parallel hash-based multi-way join [4, 29, 13], they still retrieve entire relations from the underlying KV-stores and incur heavy communication cost, the same problem that binary hash join experiences.

Parallelizing KBA plans. For a KBA plan ξ, one could follow the parallelization strategy above, *i.e.,* (a) fetch relevant KV instances in ξ from the BaaV-store $\vec{\mathcal{D}}$, (b) flatten blocks into relations and replace all $\propto$ operations in ξ as $\bowtie$, and (c) execute ξ using parallel join algorithms (cf. [29, 31, 13, 4]). While this method can be directly supported by existing SQL layers in SQL-over-NoSQL systems, it does not take advantages of BaaV and KBA plan ξ since it still retrieves all relevant relations. Hence, even when ξ is scan-free over $\vec{\mathcal{D}}$, its parallel execution still involves costly scans.

To overcome this, we propose a method to execute KBA plans ξ in parallel on BaaV $\vec{\mathcal{D}}$. It ensures the following.

(1) Bounded communication. If ξ is scan-free over $\vec{\mathcal{D}}$, the parallelized execution of ξ remains scan-free, and moreover, if $\vec{\mathcal{D}}$ has a bounded degree, then the parallel plan incurs bounded data access and bounded communication cost.

(2) Parallel scalability. The parallelized ξ is parallel scalable for both computational and communication costs.

(3) Horizontal scalability. Zidian preserves the horizontal scalability of existing SQL-over-NoSQL systems, which is measured as the system I/O throughput when adding new storage nodes. This will also be experimentally verified in Section 9.

7.2 Interleaved Parallelization

We next present our parallelization strategy, and prove its bounded communication and parallel scalability.

As opposed to fetching the entire data needed from the storage layer first and then executing ξ on the data by the computing nodes, Zidian accesses data, parallelizes plan and executes the plan in an *interleaved* manner to reduce data access and communication cost. Moreover, when ξ is scan-free, it fetches data only using $\propto$ operations, without scan.

Zidian parallelizes operations in ξ one by one, starting from leaf nodes. For an operation δ, Zidian parallelizes it as follows.

(1) δ is $\vec{R}_1 \propto \vec{R}_2$. Zidian does the following: (a) re-partition keyed blocks of $\vec{R}_1$ (intermediate results) based on key distribution of $\vec{R}_2$; this is supported by, *e.g.,* SparkSQL; (b) distribute partitions of $\vec{R}_1$ to storage nodes in which $\vec{R}_2$ resides; and (c) at each relevant storage node, retrieve only needed keyed blocks of $\vec{R}_2$ in $\vec{\mathcal{D}}$, by making use of the distributed keyed blocks of $\vec{R}_1$ as "constants". This allows us to fetch necessary data in $\vec{R}_2$ without scanning it. Note that data access and parallel execution are interleaved.

(2) δ is σ, π, $\bowtie$, $\cup$ or $-$. Zidian parallelizes δ by applying existing parallelization methods for RA operations (cf. [39, 31]). Different from (a), these do not involve data access.

(3) δ is a KV schema $\vec{R}$. Such operation does not appear in a scan-free KBA plan. If δ is $\vec{R}$, then ξ is not scan-free and it has to scan the KV instance of $\vec{R}$ from $\vec{\mathcal{D}}$. It is parallelized in the same way as SQL-over-NoSQL systems.

(4) δ is $\uparrow$. Operation $\uparrow$ is used to align KV instances for operations $\propto$, $\cup$ or $-$. It is executed along with their parallelization without changing the complexity asymptotically.

We denote by ξ_p the KBA plan ξ parallelized as above.

We next verify that the parallelized scan-free KBA plans indeed scan no table and are parallel scalable.

Communication boundedness. We have the following.

Proposition 7: *For scan-free KBA plans ξ over $\vec{\mathcal{D}}$, ξ_p (a) incurs no scans of KV instances, and (b) if ξ is bounded on $\vec{\mathcal{D}}$, then it incurs bounded communication cost.* □

Parallel scalability. To characterize the effectiveness of parallel processing, we adopt a criteria introduced by [30] that has been widely used in practice. Suppose that the computing cluster has p nodes. The complexity of a parallelized plan ξ_p over BaaV-store $\vec{\mathcal{D}}$, denoted by $T_{\mathsf{par}}(\xi_p, \vec{\mathcal{D}})$, is measured by:

$$T_{\mathsf{par}}(\xi_p, \vec{\mathcal{D}}) = T_{\mathsf{comm}}(\xi_p, \vec{\mathcal{D}}) + T_{\mathsf{comp}}(\xi_p, \vec{\mathcal{D}}),$$

where $T_{\mathsf{comm}}(\xi_p, \vec{\mathcal{D}})$ and $T_{\mathsf{comp}}(\xi_p, \vec{\mathcal{D}})$ are the communication and computation costs incurred by the execution of ξ_p on $\vec{\mathcal{D}}$ over p computing nodes, respectively. To simplify the discussion, we assume *w.l.o.g.* that $\vec{\mathcal{D}}$ is *not skewed*, *i.e.,* when parallelizing $D_1 \bowtie D_2$ using hash-join over p machines, its parallel complexity is $\frac{|D_1||D_2|}{p^2}$ [39, 31].

We say that a parallelized plan ξ_p is *parallel scalable relative to* ξ if all for BaaV stores $\vec{\mathcal{D}}$, with p computing nodes,

$$T_{\mathsf{par}}(\rho(\xi), \vec{\mathcal{D}}) = O(T_{\mathsf{seq}}(\xi, \vec{\mathcal{D}})/p),$$

where $T_{\mathsf{seq}}(\xi, \vec{\mathcal{D}})$ is the sequential complexity of ξ over $\vec{\mathcal{D}}$.

Intuitively, the parallel scalability measures speedup over sequential plan ξ by parallelization. It is relative to the yardstick plan ξ. A parallel scalable ξ_p guarantees to reduce the running time of ξ when more computing nodes are used.

Theorem 8: *When $\vec{\mathcal{D}}$ is not skewed, interleaved parallel plans ξ_p are parallel scalable relative to* KBA *plans ξ.* □

Theorem 8 shows that ξ_p is parallel scalable, no matter whether it is scan-free or not. Proposition 7 assures that ξ_p remains scan-free (resp. bounded) as long as ξ is scan-free (resp. bounded) by interleaving data access and computation. Due to the space constraint we defer the proof to [2].

Remark. While interleaved parallelization is, to some extent, in the same spirit of operation pipelining and pushdown in distributed databases [39], it is not easy to support them in the existing SQL-over-NoSQL systems under the TaaV model. Indeed, under TaaV, selection predicates and intermediate results (say, on X-attributes) cannot be efficiently used to fetch new data in an on-demand manner to avoid excessive `get` invocation and scans, since X-attributes are not a key. We view **M3** as an immediate benefit of BaaV, showcasing that with BaaV, conventional advanced parallelization methods in distributed databases can be easily adopted in SQL-over-NoSQL, which is hard, if not impossible, with TaaV.

8. SYSTEM

We show how Zidian decides the BaaV-store for module **M4** (Section 8.1) and how Zidian is implemented (Section 8.2).

8.1 From TaaV to BaaV

Given a database $\mathcal{D}$ of schema $\mathcal{R}$, Zidian first decides a BaaV schema $\vec{\mathcal{R}}$ for $\mathcal{D}$. It then maps $\mathcal{D}$ to $\vec{\mathcal{R}}$, yielding BaaV store $\vec{\mathcal{D}}$.

Zidian employs QCS from historical queries. A QCS has the form $Z[X]$, where X and Z are sets of attributes of $\mathcal{R}$ and $X \subseteq Z$. It extends *query-column-set* of *e.g.,* AQP systems [5], by distinguishing "known" attributes X from query-column-set Z to abstract the access patterns of query execution.

Intuitively, $Z[X]$ is an "access pattern" of query plans. It says that a plan often accesses attributes Z of a relation when X-values are already known. A query Q is *abstracted* by a set Σ of QCS if Q has a plan that follows the "access patterns" of QCS in Σ (see [2] for a formal definition). For example, query $Q = \pi_F(\sigma_{A=1} R(A, B, C) \bowtie_{B=E} S(E, F, G))$ could be abstracted as two QCS $\phi_1 = AB[A]$ and $\phi_2 = EF[E]$ since (a) only AB and EF attributes are needed by Q, (b) from $A = 1$ and ϕ_1 we can get related B-values and in turn, E-values by $\bowtie_{B=E}$; and (c) F-values are related to E-values by ϕ_2.

BaaV schema design. Zidian provides an algorithm (T2B) that, given a schema $\mathcal{R}$, a database $\mathcal{D}$ of $\mathcal{R}$, a set Σ of QCS, and a storage budget b, computes BaaV schema $\vec{\mathcal{R}}$ so that:

(a) the mapping $\vec{\mathcal{D}}$ of $\mathcal{D}$ on $\vec{\mathcal{R}}$ is no larger than b; and
(b) all QCS $Z[X]$ in Σ are *supported* by $\vec{\mathcal{R}}$ if $b \geq |\mathcal{D}|$ (size of $\mathcal{D}$): for any X-value, all associated Z values in the database $\mathcal{D}$ of $\mathcal{R}$ can be fetched from $\vec{\mathcal{D}}$, and even without scans when b is sufficiently large.

When b permits, $\vec{\mathcal{R}}$ is data preserving for $\mathcal{R}$ by **Condition (I)**. As an added flexibility by the implementation, Zidian also exposes an interface for the users to modify $\vec{\mathcal{R}}$ with suggested KV schemas, allowing human-in-the-loop schema design.

We outline algorithm T2B (see full version [2] for details).

(1) T2B first generates an initial BaaV schema $\vec{\mathcal{R}}_0$ by treating each QCS $Z[X]$ as a KV schema $\langle Z, X \setminus Z \rangle$. For any query Q abstracted by the QCS in Σ, Q is scan-free over $\vec{\mathcal{R}}_0$.

(2) It then removes *redundant* KV schemas in $\vec{\mathcal{R}}_0$: a KV schema $\vec{R}\langle Z, X \rangle$ is redundant in $\vec{\mathcal{R}}_0$ if for any query Q, Q is scan-free over $\vec{\mathcal{R}}_0$ *if and only if* Q is scan-free over $\vec{\mathcal{R}}_0 \setminus \{\vec{R}\}$. When there exist multiple redundant KV schemas in $\vec{\mathcal{R}}_0$, algorithm T2B removes them one by one by using a ranking function that picks the one with minimum estimated impact on the efficiency of query evaluation over $\vec{\mathcal{R}}_0$.

(3) If the mapping of $\mathcal{D}$ on the BaaV schema $\vec{\mathcal{R}}'$ generated by step (2) exceeds the budget b, T2B iteratively merges KV schemas in $\vec{\mathcal{R}}'$ to reduce the size of the mapping while keeping all scan-free queries over $\vec{\mathcal{R}}'$, until the size is within b. In each iteration, T2B picks a pair of KV schemas with minimum estimated impact on the efficiency of query evaluation.

8.2 Implementation

As a proof of concept, we have implemented Zidian. Below we outline its implementation and deployment.

Realizing BaaV. A keyed block (k, B) is realized in Zidian by encapsulating the block B of tuples as a single value. In this way, keyed blocks, and hence KV instances under BaaV, can be supported by all existing KV storage systems.

If B of a keyed block (k, B) has size above a threshold s, Zidian breaks (k, B) into multiple smaller keyed blocks such that each block does not exceed s. The decomposed blocks share the same X-values but are assigned with distinct internal ID segments appended to X; they logically appear as one keyed block. In Zidian, each relation is controlled by an individual threshold (500MB by default).

Note that conventional TaaV stores are a special case of BaaV stores since a KV instance under TaaV essentially consists of keyed blocks with block size threshold set to 1 tuple.

Remark. Modern KV storage systems support complex values and multi-map data abstraction. Because of this, keyed blocks and BaaV-stores can be supported by them without any modification. We position BaaV as an alternative model for storing relations using existing KV systems, to unleash their flexibility and performance that the TaaV model has not explored.

Modules. Zidian supports all modules (**M1** – **M4**) of Fig. 1b. More specifically, we have implemented algorithms, all in C++, for Zidian to (a) check whether $\vec{\mathcal{R}}$ is data preserving

Table 2: Case study: Q_1 of Example 3 (Exp-1)

	SoH	SoH$_{Zidian}$	SoK	SoK$_{Zidian}$	SoC	SoC$_{Zidian}$
time (s)	1.3×10^2	12.4	40.5	5.4	88.1	9.9
#data	5.2×10^8	8.4×10^6	5.2×10^8	8.4×10^6	5.2×10^8	8.4×10^6
#get	1.0×10^8	5.2×10^4	1.0×10^8	5.2×10^4	1.0×10^8	5.2×10^4
comm (MB)	4.6×10^2	16.7	4.5×10^2	15.4	4.5×10^2	15.7

for $\mathcal{R}$ (based on the proof of Theorem 1 in [2]; for **M1**); (b) check whether over the internal BaaV schema $\vec{\mathcal{R}}$, an input SQL query Q (i) can be answered and (ii) is scan-free (proofs of Theorems 2, 4 in [2]; for **M1**); (c) generate a KBA plan ξ for Q over $\vec{\mathcal{R}}$ (Section 6.2; for **M2**); (d) parallelize ξ (Section 7; proof of Theorem 8 in [2]; for **M3**); (e) generate the BaaV schema $\vec{\mathcal{R}}$ (T2B in Section 8.1 and [2]; for **M4**).

Zidian also includes a module to map database $\mathcal{D}$ of $\mathcal{R}$ to $\vec{\mathcal{R}}$ generated in **M4** (following Section 4.1), yielding BaaV-store $\vec{\mathcal{D}}$. In addition, it monitors changes to $\mathcal{D}$ and keeps $\vec{\mathcal{D}}$ up-to-date. In response to updates Δ, *e.g.,* tuple insertions and deletions to $\mathcal{D}$, Zidian incrementally updates $\vec{\mathcal{D}}$ in $O(|\Delta| \cdot \mathsf{deg}(\vec{\mathcal{D}}))$-time, *independent of* the size of $\mathcal{D}$ and $\vec{\mathcal{D}}$, where $|\Delta|$ is the size of Δ and $\mathsf{deg}(\vec{\mathcal{D}})$ is the degree of $\vec{\mathcal{D}}$ (see [2]).

These modules are all *platform independent* in the sense that their implementation works with all KV storage systems.

Deploying Zidian. To deploy Zidian for a SQL-over-NoSQL combination, an adapter is needed for Zidian to connect to the underlying KV system (*i.e.,* the NoSQL part), by carrying out KBA plans with the latter. It is *platform dependent* since it involves interactions with the KV storage.

This can typically be realized by implementing KBA operations using connectors of the KV systems for SQL layers atop them, so that we do not need to directly translate KBA plans into low-level `get` sequences. For instance, to deploy Zidian for SparkSQL-over-HBase [7], we simply extend the SparkSQL (Spark) connector of HBase to interpret KBA plans as RDD operations. Similarly, Zidian can be deployed for SparkSQL-over-Kudu [1] and SparkSQL-over-Cassandra [6].

Added functionality. Zidian extends SQL-over-NoSQL systems with unique features offered by the BaaV model.

(1) Compression. BaaV allows data compression for keyed blocks. More specifically, when mapping an instance I of schema $R(X, Y, Z)$ to a KV instance $\vec{I}$ of KV schema $\langle X, Y \rangle$ under BaaV, for each keyed block (k, B) in $\vec{I}$, B consists of *distinct* Y-values only, where each $\bar{y} \in B$ is attached with a counter c that records the multiplicity of $(k, \bar{y})$ in I.

(2) Statistics. Zidian also maintains group-by statistics aggregated over keyed blocks to speed up aggregate queries. For each keyed block (k, B) of KV schema $\langle X, Y \rangle$, it records the min, max, sum and avg of numeric attributes in B. We find such statistics useful for aggregate queries grouped by X.

9. EXPERIMENTAL STUDY

Using benchmark and real-life datasets, we conducted four sets of experiments to evaluate the effectiveness of Zidian to improve existing SQL-over-NoSQL for (1) overall performance, (2) ability to reduce scans, (3) parallel scalability and communication reduction, and (4) KV workload performance.

Experimental settings. We start with the settings.

Benchmark. We used TPC-H benchmark [42] and tested its 22 built-in benchmark queries. We generated 8 relations with 61 attributes of different scales using TPC-H dbgen.

Table 3: Average time (s): 128GB, 8 workers

	SoH	SoH$_{Zidian}$	SoK	SoK$_{Zidian}$	SoC	SoC$_{Zidian}$
MOT	3.3×10^3	1.4	4.3×10^2	0.3	7.6×10^2	0.3
AIRCA	1.0×10^3	1.1	1.2×10^2	0.4	1.8×10^3	0.4
TPC-H	1.5×10^3	96.1	1.9×10^2	52.2	3.1×10^2	1.2×10^2

Real-life datasets. We also used two real-life datasets.

(a) UK MOT data (MOT) integrates anonymized UK MOT tests data [23] and roadside survey of vehicle observations [24] on the UK road network. It has 3 tables with 42 attributes, about 16GB of data records from 2007 to 2011.

(b) US Air carriers (AIRCA) records flight statistics of US air carriers. It consists of Flight On-Time Performance Data [44] and Carrier Statistic data [45]. It has 7 tables, 358 attributes, and about 32 GB of data for records from 1987 to 2001.

To test scalability, we scaled up both real-life datasets up to 128GB by populating tuples with values from active domains.

Queries. We designed a generator to produce queries with varying structures over the two real-life datasets. We manually created 12 query templates for each dataset with 1 to 3 joins. The generator populates these templates by randomly instantiating their parameters with values from the datasets, yielding 36 queries for each real-life datasets.

BaaV *schema*. We extracted 8, 8 and 64 KV schemas for MOT, AIRCA and TPC-H queries, respectively, from QCS derived from their historical execution plans using algorithm T2B (Section 8.1; the storage budget is set to 3.5 times of the size of the datasets). Over them, for each real-life dataset, query templates $q_1 - q_6$ are scan-free and $q_7 - q_{12}$ are not; $q_1 - q_6$ are also bounded on both datasets since the relevant KV instances have stable and bounded degrees. For TPC-H queries, $q_2, q_3, q_5, q_7, q_8, q_{10}, q_{11}, q_{12}, q_{17}, q_{19}$ and q_{21} are scan-free but are not bounded since TPC-H datasets have large and varying degrees; all other TPC-H queries are not scan-free because they simply aggregate over (ranges of) entire datasets. The KV schemas are result preserving for all these queries.

The BaaV schemas and real-life queries are reported in [2].

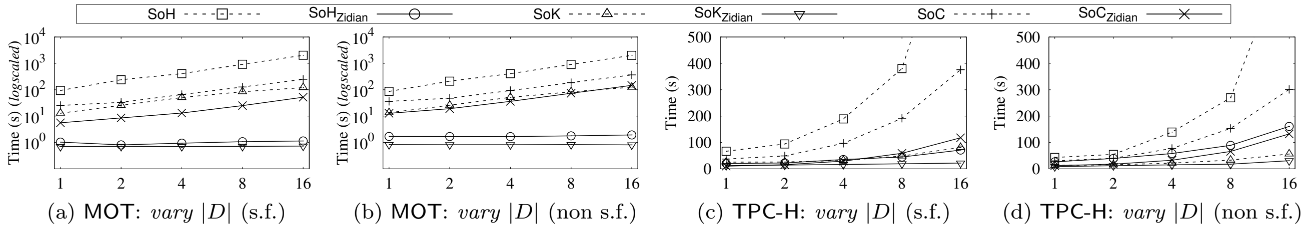
Systems. We implemented Zidian and deployed it on top of three baselines: (a) SparkSQL-over-HBase(SoH), (b) SparkSQL-over-Kudu(SoK) and (c) SparkSQL-over-Cassandra (SoC) using Spark v2.3.3, HBase v2.0.5, Kudu v1.9.0 and Cassandra v3.0.17. We denote SoH, SoK and SoC with Zidian as SoH$_{Zidian}$, SoK$_{Zidian}$ and SoC$_{Zidian}$, respectively.

Configuration. The experiments were conducted on a cluster of 12 Amazon EC2 m4.2xlarge instances, with 32GB of memory, 8 vCPUs and 500GB of SSD for each instance. Each instance works as both a computing node and a storage node, All the tests were run 3 times. The average is reported here.

Experimental Results. We next report our findings.

Exp-1: Overall performance. We first evaluated the overall performance. Using a cluster of 8 EC2 instances, we compared the (a) evaluation time (time), (b) total number of values accessed (#data), (c) number of `get` invocations (#get) and (d) size of total data shipped (comm) of all systems.

Case study. We first report the results of Q_1 of Example 3 over TPC-H of 128GB in Table 2, for which SoH$_{Zidian}$, SoK$_{Zidian}$ and SoC$_{Zidian}$ used the KBA plan generated in Example 7. Here Zidian speeds up SoH, SoK and SoC by 10.8, 7.5 and 8.8 times, respectively. With Zidian, all three systems access 62.1 times less data, invoke 2×10^3 times fewer `get` requests, and



(a) MOT: *vary* $|D|$ (s.f.) (b) MOT: *vary* $|D|$ (non s.f.) (c) TPC-H: *vary* $|D|$ (s.f.) (d) TPC-H: *vary* $|D|$ (non s.f.)

Figure 3: Impact of scans: 1 worker (Exp-2)

incur about 28 times less communication than without Zidian. This verifies the effectiveness of BaaV and Zidian.

Overall results. The evaluation time on TPC-H of 128GB and the expanded MOT and AIRCA (128GB) are reported in Table 3 (see Tables 4–15 in [2] for evaluation time, #data, #get and comm of each individual query on all datasets).

(1) $\text{SoH}_{\text{Zidian}}$, $\text{SoK}_{\text{Zidian}}$ and $\text{SoC}_{\text{Zidian}}$ outperform SoH, SoK and SoC, respectively, for each and every query on all the datasets. The speedup ratio of $\text{SoH}_{\text{Zidian}}$ over SoH is 2.9×10^3, 1.0×10^3 and 46.1 times on MOT, AIRCA and TPC-H on average, respectively, up to 4.1×10^3, 1.3×10^3 and 5.3×10^2 times. Similarly, on average Zidian improves SoK by 2.0×10^3, 3.3×10^2 and 11.6 times, up to 3.7×10^3, 5.1×10^2 and 1.1×10^2 times, respectively; over SoC the speedup ratio on average is 3.0×10^3, 5.4×10^3 and 11.6 times, up to 5.7×10^3, 9.3×10^3 and 24.1 times.

(2) For scan-free (bounded) queries for MOT and AIRCA, on average Zidian improves SoH by 2.9×10^3 and 1.1×10^3 times, respectively, up to 4.1×10^3 and 1.3×10^3 times; similarly, it speeds up SoK by 2.1×10^3 and 3.5×10^2 times on average, up to 3.7×10^3 and 5.1×10^2 times, and improves SoC by 4.3×10^3 and 7.4×10^3 times, up to 5.7×10^3 and 9.3×10^3 times.

For scan-free (unbounded) TPC-H queries, Zidian speeds up SoH, SoK and SoC by 71.4, 17.4 and 6.1 times on average, respectively, up to 5.6×10^2, 1.1×10^2 and 24.1 times.

For non scan-free MOT, AIRCA and TPC-H queries, the average speedup over SoH is 2.8×10^3, 8.6×10^2 and 20.9 times, respectively, up to 3.8×10^3, 1.2×10^3 and 1.1×10^2 times; the numbers for SoK (resp. SoC) are 2.0×10^3, 3.1×10^2, 5.8 (resp. 1.7×10^3, 3.4×10^3, 5.5) times on average, up to 3.7×10^3, 4.4×10^2, 24.6 (resp. 2.1×10^3, 5.4×10^3, 21.4) times.

Observation. The speedup on scan-free TPC-H queries is not as substantial as on MOT and AIRCA. This is because TPC-H generates skew-free datasets using uniform data distributions [21], and most of their attribute values are distinct. As a consequence, the degrees of KV instances (Section 4) under BaaV are mostly either 1 (the same as under TaaV) or almost as large as the relation size. Thus KBA plans over such skewless TPC-H datasets under BaaV are often similar to plans under TaaV. In contrast, both real-life datasets MOT and AIRCA are quite skewed. Hence, under BaaV, KV instances for MOT and AIRCA have reasonable degrees and benefit better from KBA plans. Moreover, many attributes of MOT and AIRCA have small active domains due to skewness, on which the compression and statistic features of Zidian are more effective.

Exp-2: Scan-free evaluation. To see why Zidian improves the performance, we studied the impact of scans. Using 1 EC2 instance (to exclude the impact of communication) and varying the datasets from 1GB to 16GB (we varied the size of MOT and AIRCA by partitioning on date attributes), we tested the evaluation time of all systems.

The results for scan-free (*s.f.*) and non scan-free (*non s.f.*) MOT queries are reported in Figures 3a and 3b, respectively; the results for non scan-free and scan-free TPC-H (all are unbounded) are in Figures 3c and 3d, respectively. See Figure 6 in [2] for AIRCA results (similar to the results on MOT).

(1) $\text{SoH}_{\text{Zidian}}$ is 2.0×10^2 and 1.4×10^2 times faster than SoH for bounded and non scan-free MOT queries, respectively; and the speedup is 1.1×10^2 and 88.5 (resp. 2.2×10^3 and 9.3×10^2) times over SoK (resp. SoC), respectively. For scan-free but unbounded and non scan-free TPC-H queries on 16GB of data, the speedup is 39.0 and 9.4 times over SoH, 5.9 and 3.0 times over SoK, and 5.3 and 3.0 times over SoC, respectively. Zidian improves scan-free queries better than non scan-free ones by avoiding scans. It also improves non scan-free queries as they contain many scan-free subqueries. These verify that scan inflicts a large cost on query time for SQL-over-NoSQL and Zidian effectively improves that with scan-free (sub)queries. Note that HBase (SoH) is the slowest among the three.

(2) The evaluation time of bounded queries by $\text{SoH}_{\text{Zidian}}$ is indifferent to $|D|$: 0.7 seconds on 1GB and 0.7 seconds on 16GB of MOT, while SoH increases from 12.9 seconds to 1.2×10^2 seconds; similarly for $\text{SoK}_{\text{Zidian}}$ and $\text{SoC}_{\text{Zidian}}$. This verifies that with BaaV, bounded queries can be answered by using a bounded amount of data regardless of $|D|$.

Exp-3: Parallel scalability and communication cost. We also find that Zidian guarantees parallel scalability and reduces the communication cost of SQL-over-NoSQL systems, in addition to efficiency improvement with the BaaV model.

Varying p. Using 32 GB of each of MOT, AIRCA and TPC-H, we varied the number p of workers (EC2 instances) from 4 to 12, and tested the evaluation time and total data shipment of all systems. The results over MOT and TPC-H are shown in Figures 4a–4d (see Figures 7a–7b in [2] for AIRCA results).

(1) Zidian consistently speeds up SoH, SoK and SoC in all cases. On average, $\text{SoH}_{\text{Zidian}}$ is 5.5×10^2, 1.6×10^2 and 10.9 times faster than SoH over MOT, AIRCA and TPC-H, respectively, up to 1.2×10^3, 3.6×10^2 and 1.9×10^2 times. The speedup is 6.3×10^2, 95.3 and 5.5 on average for SoK, up to 1.9×10^3, 2.1×10^2 and 45.7 times respectively, and 1.0×10^3, 1.2×10^3 and 4.9 times for SoC, up to 2.5×10^3, 2.4×10^3 and 15.5 times, respectively.

(2) Moreover, Zidian reduces total communication cost of SoH, SoK and SoC. The total communication of $\text{SoH}_{\text{Zidian}}$ accounts for 0.03%, 0.15% and 22.7% of that of SoH over MOT, AIRCA and TPC-H, respectively; similarly for SoK and SoC.

(3) Zidian does not hamper the scalability of SoH, SoK and SoC: they all scale well with p. Varying p from 4 to 12, the evaluation time of $\text{SoH}_{\text{Zidian}}$ improves by 2.0, 2.5 and 2.2 times on MOT, AIRCA and TPC-H, respectively, while it is 2.5, 2.8 and 2.6 times for SoH. The improvements for $\text{SoK}_{\text{Zidian}}$ vs. SoK and $\text{SoC}_{\text{Zidian}}$ vs. SoC are similar. The running time of

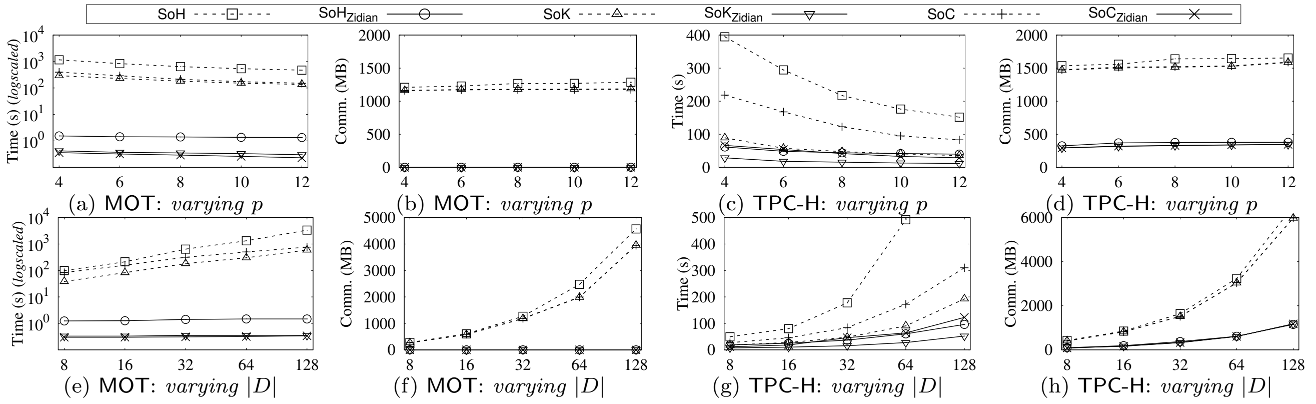


Figure 4: Parallel scalability and communication cost (Exp-3)

SoH improves slightly better than with Zidian when increasing p since the evaluation time of SoH$_{Zidian}$ is much smaller and hence system initialization overhead of its underlying SoH plays a larger role in the performance of SoH$_{Zidian}$ when adding workers; similarly for SoK and SoC.

Varying $|D|$. Fixing $p = 8$, we varied the size $|D|$ of the datasets from 8GB to 128GB, and evaluated all systems. The results over MOT and TPC-H are reported in Figures 4e–4h (see Figures 7c–7d in [2] for results over AIRCA).

(1) Consistent with the results above, Zidian improves SoH, SoK, SoC in efficiency and communication cost in all cases.

(2) All systems, with or without Zidian, scale well with $|D|$. On average SoH$_{Zidian}$ takes 1.4, 1.1 and 96.1 seconds for queries on 128GB of MOT, AIRCA and TPC-H, respectively. In contrast, SoH takes $3.4{\times}10^3$, $1.0{\times}10^3$ and $1.6{\times}10^3$ seconds in the same setting; similarly for SoK$_{Zidian}$ vs. SoK and SoC$_{Zidian}$ vs. SoC. Moreover, with Zidian the evaluation time of scan-free queries of all systems decreases slower than that of non scan-free queries, justifying the benefit of BaaV and scan-free queries.

(3) The communication of SoH$_{Zidian}$ for bounded queries is consistently about 0.33MB and 0.25MB, respectively, for bounded MOT and AIRCA queries when $|D|$ varies from 8GB to 128GB, while SoH increases from $2.8{\times}10^2$MB to $4.6{\times}10^3$MB and from 45.9MB to $8.5{\times}10^2$MB. Similarly, SoK$_{Zidian}$ (resp. SoC$_{Zidian}$) incurs stable communication.

Exp-4: Support for KV workload. We next evaluated the impact of BaaV and Zidian on SoH, SoK and SoC for executing key-value workload: throughput and horizontal scalability.

Throughput. We evaluated the throughput of all systems in the same setting as Exp-1, which is measured by TPMS: the number of values processed per ms by all workers. We did not use # of gets/puts processed because a get under BaaV retrieves values involving multiple gets under TaaV.

For read workload (bulk gets), we found that Zidian improves the average TPMS of SoH by 1.5, 1.4 and 1.1 times over MOT, AIRCA and TPC-H, respectively. For write workload (bulk puts), TPMS of SoH$_{Zidian}$ is 67.4%, 74.1% and 90.1% of that of SoH on MOT, AIRCA and TPC-H, respectively; similarly for SoK and SoC. Zidian improves read throughput because get under BaaV is more efficient than under TaaV with more efficient get invocations. However, a put with key-value pair (k, v) under BaaV has to process more values than under TaaV when k already exists in the storage; so its write throughput is a bit lower but is still comparable.

Horizontal scalability. Fixing the size of data at each worker to 10 GB, we varied the number of workers from 4 to 12, and tested TPMS of read and write workloads as above (see Figure 8 in [2] for results). The average throughput (TPMS) of both SoH and SoH$_{Zidian}$ increases almost linearly, from $5.8{\times}10^2$ and $7.7{\times}10^2$ to $1.5{\times}10^3$ and $1.9{\times}10^3$ for read, and from $2.1{\times}10^2$ and $1.6{\times}10^2$ to $5.3{\times}10^2$ and $4.1{\times}10^2$ for write, respectively; similarly for SoK$_{Zidian}$ and SoC$_{Zidian}$. These verify that Zidian retains the horizontal scalability of SoH, SoK and SoC.

Summary. We find the following. (1) On average Zidian outperforms SoH, SoK and SoC in efficiency by $2.8{\times}10^2$, $1.7{\times}10^2$ and $8.1{\times}10^2$ times for scan-free queries, respectively, and by $2.0{\times}10^2$, $1.5{\times}10^2$ and $3.6{\times}10^2$ times for non scan-free queries. Moreover, on average it reduces communication cost of SoH, SoK and SoC by $1.5{\times}10^3$, $2.9{\times}10^3$ and $4.2{\times}10^3$ times. (2) For bounded queries, both computation cost and communication cost of all systems with Zidian remain stable when datasets get larger. (3) Zidian is parallel scalable and scales well with datasets, *e.g.,* on average SoH$_{Zidian}$ takes 27.7 and 65.4 seconds for scan-free and non scan-free queries on datasets of 128GB over 8 workers, respectively, compared to $1.7{\times}10^3$ and $2.1{\times}10^3$ seconds by SoH. (4) Zidian retains the throughput and horizontal scalability of SoH, SoK and SoC.

10. CONCLUSION

We have proposed Zidian, a middleware to improve existing SQL-over-NoSQL systems by speeding up SQL query answering. The novelty of the work consists of (a) BaaV, a keyed-block model, and KBA, an extension of relational algebra to BaaV, to reduce blind scans and communication costs; (b) characterizations of result-preserving, scan-free and bounded KBA queries; (c) algorithms for generating (scan-free) query plans; and (d) a parallelization strategy that guarantees the parallel scalability of parallel query processing, preserves the scan-free (resp. bounded) property of scan-free (resp. bounded) KBA queries, and retains the horizontal scalability of SQL-over-NoSQL systems. Our experimental study has verified that Zidian substantially outperforms existing SQL-over-NoSQL systems in SQL query answering.

Acknowledgments. The authors are supported by ERC 652976, NSFC 61421003, EPSRC EP/M025268/1, Shenzhen Institute of Computing Sciences, and Beijing Advanced Innovation Center for Big Data and Brain Computing. Fan is also supported by Royal Society Wolfson Research Merit Award WRM/R1/180014 and Edinburgh-Huawei Joint Lab.

11. REFERENCES

[1] Apache Kudu. *https://kudu.apache.org/*.

[2] Full version. *http://homepages.inf.ed.ac.uk/ycao/BEKVfull.pdf*.

[3] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[4] F. N. Afrati and J. D. Ullman. Optimizing joins in a Map-Reduce environment. In *EDBT*, 2010.

[5] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[6] Apache. Cassandra. *http://cassandra.apache.org/*.

[7] Apache. Hbase. *https://hbase.apache.org/*.

[8] Apache. Hive. *https://hive.apache.org/*.

[9] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. SCADS: Scale-independent storage for social computing applications. In *CIDR*, 2009.

[10] M. Armbrust, S. Tu, A. Fox, M. J. Franklin, D. A. Patterson, N. Lanham, B. Trushkowsky, and J. Trutna. PIQL: A performance insightful query language. In *SIGMOD*, 2010.

[11] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in Spark. In *SIGMOD*, 2015.

[12] D. F. Bacon, N. Bales, N. Bruno, B. F. Cooper, A. Dickinson, A. Fikes, C. Fraser, A. Gubarev, M. Joshi, E. Kogan, A. Lloyd, S. Melnik, R. Rao, D. Shue, C. Taylor, M. van der Holst, and D. Woodford. Spanner: Becoming a SQL system. In *SIGMOD*, 2017.

[13] P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. *J. ACM*, 64(6), 2017.

[14] M. Benedikt, B. ten Cate, and E. Tsamoura. Generating low-cost plans from proofs. In *PODS*, 2014.

[15] L. Braun, T. Etter, G. Gasparis, M. Kaufmann, D. Kossmann, D. Widmer, A. Avitzur, A. Iliopoulos, E. Levy, and N. Liang. Analytics in motion: High performance event-processing AND real-time analytics in the same database. In *SIGMOD*, 2015.

[16] Y. Cao and W. Fan. An effective syntax for bounded relational queries. In *SIGMOD*, 2016.

[17] Y. Cao and W. Fan. Data driven approximation with bounded resources. *PVLDB*, 10(9):973–984, 2017.

[18] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[19] Cockroach Labs. Cockroachdb. *https://www.cockroachlabs.com/*.

[20] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally distributed database. *TODS*, 31(3), 2013.

[21] A. Crolotte and A. Ghazal. Introducing skew into the TPC-H benchmark. In *TPCTC*, 2011.

[22] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[23] Department for Transport. Anonymised mot tests and results. *http://data.gov.uk/dataset/anonymised_mot_test*.

[24] Department for Transport. Roadside survey of vehicle observations. *https://data.gov.uk/dataset/52e1e2ab-5687-489b-a4d8-b207cd5d6767/roadside-survey-of-vehicle-observations*.

[25] Facebook. MyRocks. *https://code.facebook.com/posts/190251048047090/myrocks-a-space-and-write-optimized-mysql-database/*.

[26] W. Fan, F. Geerts, Y. Cao, and T. Deng. Querying big data by accessing small data. In *PODS*, 2015.

[27] W. Fan, F. Geerts, and L. Libkin. On scale independence for querying big data. In *PODS*, 2014.

[28] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[29] P. Koutris, S. Salihoglu, and D. Suciu. Algorithmic aspects of parallel data processing. *Foundations and Trends in Databases*, 8(4), 2018.

[30] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *TCS*, 71(1):95–132, 1990.

[31] R. Li, M. Riedewald, and X. Deng. Submodularity of distributed join computation. In *SIGMOD*, pages 1237–1252, 2018.

[32] Z. Liu and S. Idreos. Main memory adaptive denormalization. In *SIGMOD*, 2016.

[33] S. Loesing, M. Pilman, T. Etter, and D. Kossmann. On the design and scalability of distributed shared-data databases. In *SIGMOD*, 2015.

[34] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at Facebook. In *NSDI*, 2013.

[35] NuoDB. NuoDB. *https://www.nuodb.com/*.

[36] J. Petit, F. Toumani, J. Boulicaut, and J. Kouloumdjian. Towards the reverse engineering of denormalized relational databases. In *ICDE*, 1996.

[37] M. Pilman, K. Bocksrocker, L. Braun, R. Marroquin, and D. Kossmann. Fast scans on key-value stores. *PVLDB*, 10(11):1526–1537, 2017.

[38] M. A. Qader, S. Cheng, and V. Hristidis. A comparative study of secondary indexing techniques in lsm-based NoSQL databases. In *SIGMOD*, 2018.

[39] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2000.

[40] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed SQL

database that scales. *PVLDB*, 6(11):1068–1079, 2013.
[41] J. Tatemura, O. Po, W. Hsiung, and H. Hacigümüs. Partiqle: An elastic SQL engine over key-value stores. In *SIGMOD*, 2012.
[42] TPC. TPC-H. *http://www.tpc.org/tpch/*.
[43] Unisphere Research. The 2016 enterprise data management survey. *http://www.unisphereresearch.com/Content/ReportDetail.aspx?IssueID=6559*.
[44] United States Department of Transportation. Airline on-time performance data. *http://www.transtats.bts.gov/DatabaseInfo.asp?DB_ID=120*.
[45] United States Department of Transportation. Statistical products and data. *http://www.transtats.bts.gov/DatabaseInfo.asp?DB_ID=110*.