Are Joins over LSM-trees Ready: Take RocksDB as an example. [Technical Report]

Weiping Yu*
Fan Wang*
weiping001@e.ntu.edu.sg
FAN008@e.ntu.edu.sg
Nanyang Technological University

Xuwei Zhang
Nanyang Technological University
zhan0612@e.ntu.edu.sg

Siqiang Luo
Nanyang Technological University
siqiang.luo@ntu.edu.sg

ABSTRACT

LSM-tree-based data stores are widely adopted in industries for their excellent performance. As data scales increase, disk-based join operations become indispensable yet costly for the database, making the selection of suitable join methods crucial for system optimization. Current LSM-based stores generally adhere to conventional relational database practices and support only a limited number of join methods. However, the LSM-tree delivers distinct read and write efficiency compared to the relational databases, which could accordingly impact the performance of various join methods. Therefore, it is necessary to reconsider the selection of join methods in this context to fully explore the potential of various join algorithms and index designs. In this work, we present a systematic study and an exhaustive benchmark for joins over LSM-trees, using RocksDB, a widely adopted LSM-based store, as a representative example. We define a configuration space for join methods, encompassing various join algorithms, secondary index types, and consistency strategies. Additionally, we summarize a theoretical analysis to assess the overhead of each join method for an in-depth understanding of their performance. Finally, we implement all join methods in the configuration space on a unified platform and compare their performance through extensive experiments. Our theoretical and experimental results reveal that no single join method dominates across all scenarios. Instead, factors such as entry size, data distribution, and join frequency should be carefully considered when selecting join methods for LSM-trees. We also offer additional useful insights and takeaways to help developers tailor join methods to their specific working conditions.

PVLDB Reference Format:
Weiping Yu, Fan Wang, Xuwei Zhang, and Siqiang Luo. Are Joins over LSM-trees Ready: Take RocksDB as an example.
[Technical Report]. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/we1pingyu/lsmjoin.

*Both authors contributed equally to this research.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

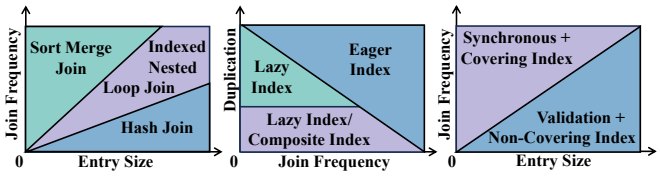

Figure 1: Our benchmark provides helpful guidelines to assist practitioner in selecting proper methods according to their working conditions for join in LSM-based key-value stores.

1 INTRODUCTION

Log-structured merge (LSM) trees [61] based key-value stores have gained significant traction in the industry. Notable examples include RocksDB [24] at Facebook, LevelDB[27] and BigTable [13] at Google, HBase [1] and Cassandra [5] at Apache, X-Engine [29] at Alibaba, WiredTiger [2] at MongoDB, and Dynamo [22] at Amazon. These LSM-based key-value stores play crucial roles in various applications such as social media [6, 10], stream processing [12, 15], and file systems [36, 68].

Joins over LSM-based stores. Many renowned LSM-based stores have implemented join operations [5, 11, 22, 24, 56, 70] as shown in Table 1. However, they typically rely on insights from relational databases to focus on a few join methods. This raises the question: Are these methods still the most appropriate choice for LSM-based stores? Our analysis suggests a likely negative answer due to the distinct and complex impact of LSM-trees. Specifically, while LSM-tree enhances updates and lookups efficiency, it requires additional costs to maintain consistency, and the overall impact of LSM-tree has not been meticulously evaluated. Consequently, there may exist substantial space for optimizing the join method selection strategy in LSM-based stores due to two problems.

Problem 1: Many join methods remain unexplored in existing LSM-based stores. As indicated in Table 1, current LSM-based databases only practice limited join methods. However, the potential of other join techniques has not been fully explored. For instance, most LSM-based stores employ a synchronous consistency strategy instead of a validation one for handling LSM-trees' out-of-place updates. This rationale is supported by the synchronous strategy's advantage in join efficiency, as the validation strategy incurs additional joining overhead. However, synchronous strategy simultaneously introduces substantial update overhead which can conversely deteriorate the performance under certain conditions. In such cases, the validation strategy may surprisingly provide better

Table 1: Different join techniques supported in different renowned LSM-based key-value databases.

NoSQL Storage Systems	Secondary Index	Join Algorithm	Consistency Strategy	Covering Index
AsterixDB [4]	Embedded Index, Composite Keys	INLJ, SJ, HJ	Synchronous	✓
Cassandra [5]	Lazy Index	×	Synchronous	×
CockroachDB [70], PolarDB-X Engine [11]	Composite Index	INLJ, SJ, HJ	Synchronous	✓
HBase [1], LevelDB [27], RocksDB [24]	×	×	×	×
MyRocks [56]	Composite Index	INLJ	Synchronous	✓

performance. Thus, it is crucial to expand the join design space to include more join methods to enhance the selection criteria.

Problem 2: Many influential factors, tied to LSM-tree properties, on join performance have not been fully examined. While existing LSM-based stores typically select join algorithms based on selectivity inspired by relational database practices [41, 69, 72, 74], LSM-tree storage introduces additional performance considerations. For instance, the performance of indexed nested loop join (INLJ) maintains robustness for large entry sizes due to the lookup optimization techniques in LSM-tree (e.g. Bloom filters). In contrast, hash join (HJ) and sort-merge join (SJ) incur obviously increasing join cost as entry size grows. Hence, though INLJ is generally less preferred in high-selectivity cases, it still can outperform HJ and SJ when handling large entry sizes. Therefore, selecting an appropriate join method in LSM-based stores is complex and requires examining more influencing factors.

To address these problems, we present a systematic study of joins in LSM-based stores, offering an exhaustive benchmark that yields interesting new insights. Our benchmark contributes mainly to the following aspects:

We identify key join method characteristics and propose an inclusive configuration space covering both existing and potential new combinations. This configuration space includes four primitive join components, join algorithm (i.e. indexed nested loop join, sort-merge join, and hash join), secondary index (i.e. eager index, lazy index, and composite index), consistency strategy (i.e. synchronous strategy and validation strategy), and covering index (i.e. covering and non-covering). It allows us to describe existing join methods and discover novel ones, such as the indexed nested loop join combined with a non-covering eager index and validation strategy. Such an integrated study allows us to explore the join method overlooked by existing literature, which typically concentrates on certain aspects of the entire space [45, 64, 71].

We tailor the theoretical analysis to the joins over LSM-tree encompassing 3 join algorithms across 12 scenarios and 6 index designs, many of which have not been previously analyzed in existing works. Different from existing join analysis, our assessment incorporates the unique characteristics and performance of LSM-trees to evaluate the overhead of join methods. Additionally, we analyze the integrated cost of each join method in our configuration space, considering its distinct join algorithm, secondary index type, consistency strategy, and covering index. Many combinations, such as the integration of eager index designs with validation strategies, have not been explored in prior research. This thorough analysis not only leads to an in-depth understanding of the features of each join method but also guides the design of our experiments.

We implement all 29 join methods within our configuration space on a unified platform and examine them under diverse working conditions to derive guidelines for join method selection in LSM-based stores. We consider 10 distinct factors concerning the workload and LSM-tree configuration to examine their impact on the performance of diverse join methods with extensive experiments. Excitingly, this leads to several useful guidelines on join method selection, as Figure 1 illustrates. Notably, some of them are different from traditional relational databases and other data storage systems like those based on B+ trees. For instance, while relational databases typically select join algorithms based on selectivity, our findings indicate that join frequency and entry size also play crucial roles. In the *Movie* dataset with moderate selectivity, SJ is preferred if only selectivity is considered. However, when the entry size increases to 4096 bytes, INLJ can outperform both SJ and HJ by about 70% in latency, making it the most suitable join algorithm. Additionally, several overlooked secondary index designs could deliver competitive performance in certain scenarios. For instance, eager index is rarely adopted by existing works due to its perceived high construction costs. Whereas, it can outperform other indexes in frequent-join workloads. In *User* dataset, INLJ with a prevalent composite index performs better than eager index when the join frequency is less than once per 10 million updates. However, when join frequency increases to 32, eager index outperforms composite index by more than 30% in latency. The insights are thoroughly detailed in our experiment section.

The remainder of this paper is organized as follows: Section 2 provides preliminary knowledge about LSM-trees and joins in LSM-based stores; Section 3 introduces our configuration space; the experimental results and discussion are elucidated in Section 4; Section 5 summarizes the important insights and takeaways to enhance our understanding of this topic, as well as points out some directions to guide future research.

2 BACKGROUND

This section discusses background knowledge about joins over LSM-based databases. Frequently used notations are listed in Table 2.

2.1 Log-Structured Merge Trees

An LSM-tree organizes data using an in-memory write buffer and multiple on-disk levels with capacities increasing exponentially by a size ratio T. To store N entries of size e, an LSM-tree should incorporate $L = \log_T \frac{N \cdot e}{M}$ levels, where M is the write buffer size. LSM-tree supports mainly three types of operations as follows.

Updates. LSM-tree uses an out-of-place update strategy where key-value pairs, or *entries*, are initially stored in the write buffer. When this buffer is full, the entries are flushed to disk, progressing

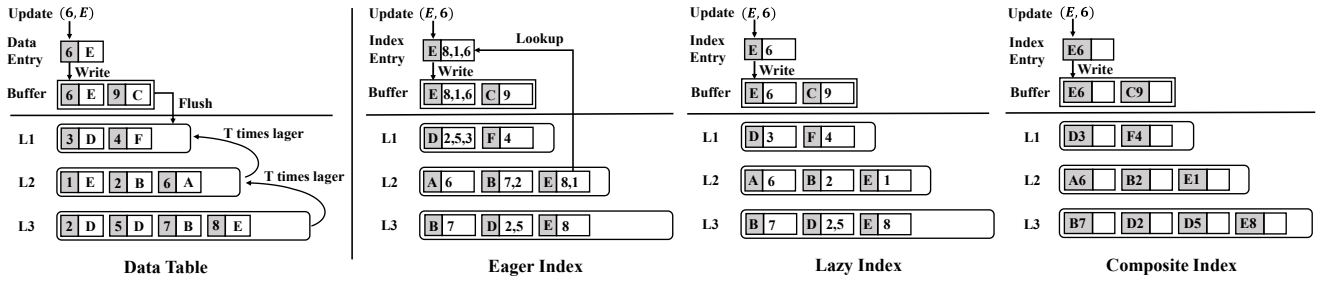

Figure 2: An illustration of LSM-trees for a data table and the corresponding index table, constructed using various secondary index types. In this figure, dark blocks represent the keys of entries, while white blocks represent the values.

Table 2: Notations used in this paper. Subscripts denote associated tables (e.g., L_R: the number of levels in right data table LSM-tree, $L_{R'}$: the number of levels in right index LSM-tree).

Term	Definition	Unit
R, S	Right and left data tables	
R', S'	Index tables for right and left data tables	
L	Number of levels in an LSM-trees	levels
N	Number of entries stored in an LSM-tree	entries
e	Size of entries of LSM-tree	bytes
B	Size of data block	bytes
p	False positive rate of Bloom filters	
ϵ	Ratio between the number of matched entries and total entries in the LSM-tree	
d	Average duplication frequency of join attribute value in a sequence of updates	entries

through the LSM-tree levels as shown in Figure 2. In the worst case, an entry reaches the largest level after $L \cdot T$ compaction processes, resulting in an update cost of $O(L \cdot T \cdot \frac{e}{B})$[18, 19].

Point Lookups. A point lookup in an LSM-tree searches for an *entry* using a given *key*. It checks through all levels, returning the result once the matched entry is found. To speed up this process, Bloom filters [9] are employed to indicate the presence of the matched entry with false positive rate p. Thus the lookup cost is $O(Lp)$ if the entry is absent, or $O(\lceil \frac{e}{B} \rceil + Lp)$ if present.

Range Lookups. A range lookup retrieves entries within a key range in an LSM-tree. It incurs $O(L)$ I/O cost to seek qualified entries across all levels [18, 19], and requires an additional $O(\frac{d \cdot e}{B})$ cost for entry retrieval, where d is the number of matched entries.

2.2 Joins in LSM-tree based databases

Joins in LSM-tree based stores typically fall into two scenarios based on the alignment of the join attribute with the primary keys of the involved data tables. We examine an elementary case involving two *data tables*, where the left and right tables are denoted as R and S. The data tables are stored in separate LSM-trees with their primary keys as the keys in LSM-tree entries. We represent these tables as $R(p_r, n_r)$ and $S(p_s, n_s)$, where p is the primary key and n represents non-primary attributes.

Hence, there are typically two possible scenarios for the data tables in joins over LSM-trees. The first scenario occurs when the join attribute is the primary key of the data table, thus entries with specific join attributes can be accessed directly via the LSM key (referred to as primary index) without additional overhead. The second, more complex scenario arises when the join attribute is a non-primary attribute. As a result, entries are difficult to identify by the join attribute since non-primary attributes are stored in the values of the LSM-tree entries. In such cases, secondary indexes can be introduced to identify and locate the associated entries, and the corresponding *index tables* are denoted as R' and S', respectively. This approach benefits the join overhead for certain algorithms (e.g., indexed nested loop joins) while incurring additional costs in index construction, thus leading to a tradeoff. Our benchmark includes both of these scenarios as well as various secondary index types to provide a comprehensive analysis.

3 CONSOLIDATED JOIN CONFIGURATION SPACE FOR LSM-TREE

Our investigation reveals that existing renowned LSM-based stores support only a limited number of join methods, leaving many potentially high-performing methods unexplored. This gap indicates a lack of systematic study of join methods in LSM-based stores, for which a comprehensive configuration space encompassing a wide range of instances is necessary.

To fill this gap we are the first to propose an inclusive configuration space tailored for joins over LSM-trees comprising four fundamental components: join algorithm, secondary index, consistency strategy, and covering index. This configuration encompasses all the existing join methods and many combinations that have never been discussed before. Additionally, we provide a theoretical analysis tailored to LSM-trees for each join method within the configuration space, which encourages a more comprehensive and thorough understanding of joins over LSM-trees.

This section provides a general introduction to and theoretical analysis of various index designs (i.e., index types, consistency strategies, and covering indexes) and join algorithms.

3.1 Secondary index types analysis

In LSM-based key-value stores, secondary indexes, which are maintained in separate LSM-trees, play a critical role in join operations. These indexes enhance join efficiency by speeding up the location

Table 3: The theoretical analysis of various secondary indexes and consistency strategies. In this table, the point lookup cost refers to the I/O cost of finding index entries associated with a certain join attribute value. Moreover, it is worth mentioning that we only present the cost for updating the index table given the update cost of the data table is identical for each method.

Index	Consistency strategy	Index Type	Empty Point Lookup (Z_0)	Non-empty Point Lookup (Z_1)	Update (U)
S-Eager	Synchronous	Eager Index	$O(L' \cdot p)$	$O(L' \cdot p + \lceil \frac{e'}{B} \rceil)$	$O(L \cdot p + \lceil \frac{e}{B} \rceil) + O(L' \cdot p + \lceil \frac{e'}{B} \rceil) + O(L' \cdot T \cdot \frac{e'}{B})$
S-Lazy		Lazy Index	$O(L' \cdot p)$	$O(L' \cdot \lceil \frac{e'}{B} \rceil)$	$O(L \cdot p + \lceil \frac{e}{B} \rceil) + O(L' \cdot T \cdot \frac{e'}{B})$
S-Comp		Composite Keys	$O(L' \cdot p)$	$O(L' + d \cdot \frac{e'}{B})$	$O(L \cdot p + \lceil \frac{e}{B} \rceil) + O(L' \cdot T \cdot \frac{e'}{B})$
V-Eager	Validation	Eager Index	$O(L' \cdot p)$	$O(d \cdot (L \cdot p + \lceil \frac{e}{B} \rceil)) + O(L' \cdot p + \lceil \frac{e'}{B} \rceil)$	$O(L' \cdot p + \lceil \frac{e}{B} \rceil)) + O(L' \cdot T \cdot \frac{e'}{B})$
V-Lazy		Lazy Index	$O(L' \cdot p)$	$O(d \cdot (L \cdot p + \lceil \frac{e}{B} \rceil)) + O(L' \cdot \lceil \frac{e'}{B} \rceil)$	$O(L' \cdot T \cdot \frac{e'}{B})$
V-Comp		Composite Keys	$O(L' \cdot p)$	$O(d \cdot (L \cdot p + \lceil \frac{e}{B} \rceil)) + O(L' + d \cdot \frac{e'}{B})$	$O(L' \cdot T \cdot \frac{e'}{B})$

and sequential extraction of data based on the join attribute, albeit at the cost of increased update overhead and space consumption, presenting a novel trade-off requiring detailed analysis. We refer to entries and keys in data and index tables as *data entries*, *index entries*, *data keys*, and *index keys*. We consider three secondary index types [45, 64]: *Eager Index*, *Lazy Index*, and *Composite Index*. Our analysis in Table 3 quantitatively evaluates their distinct performance to guide the index selection for specific conditions.

Eager Index (Eager). Eager Index is a typical LSM-style secondary index. It uses join attributes as keys and stores associated primary keys and attributes in a posting list as the value of the index entry. This structure links multiple entries that share the same join attribute value. Therefore, a multi-step process is involved to update the index LSM-tree: searching, entry creation, and writing. When an update involves a specific join attribute value, the system first searches and retrieves the index entry with the associated key by a point lookup operation, incurring an I/O cost of $O(L \cdot p + \lceil \frac{e}{B} \rceil)$. Then, create a new index entry with the same key to append the update information to the value of the retrieved entry. This generated entry is then inserted into the write buffer and subsequently compacted to larger levels at a cost of $O(L \cdot T \cdot \frac{e}{B})$. If no existing entry is found, a new entry is directly created to store the update information. For example, as shown in Figure 2, an update of index LSM-tree is triggered by a data table update $(6, E)$, where 6 is the primary key and E is the join attribute. The system searches the index LSM-tree to locate the existing entry with the key E at level-2. Then, merge it with the update information $(E, 5)$ to form the new entry $\{E|8, 1, 6\}$, which is then written to the write buffer.

Once the index LSM-tree is built, data associated with specific join attributes can be efficiently extracted through point lookups. According to Section 2.1, the cost of data extraction via point lookups is $O(L \cdot p)$ for empty lookups and $O(L \cdot p + \lceil \frac{e}{B} \rceil)$ for non-empty ones. Given the typically low false positive rate p of Bloom filters, the cost of non-empty lookups is obviously higher than that of empty lookups, potentially favoring the eager index and indexed nested loop joins for certain workloads, which will be further discussed in the evaluation section.

Lazy Index (Lazy). Lazy Index also uses posting lists to store associated primary keys within an index entry but defers their merging. Unlike Eager Index, it merely appends a new entry to the write buffer, with index entries gradually merging during compaction at a cost of $O(L \cdot T \cdot \frac{e}{B})$. As a result, entries with the same key may exist concurrently at different LSM-tree levels, requiring a modified lookup process. Instead of stopping after finding the first matching entry, the process must examine all levels to collect all relevant entries. Given that the entries are stored in sorted run in each level, there is at most one matching index entry per level, resulting in a lookup cost of $O(L \cdot \lceil \frac{e}{B} \rceil)$. For instance, as Figure 2 shows, retrieving entries with join attribute E requires extracting index entries from multiple levels to compile the complete posting list $\{6, 1, 8\}$. Lazy Index provides better update efficiency but incurs higher lookup overhead, suiting workloads requiring rapid index updates but tolerating infrequent point lookups.

Composite Index (Comp). The secondary index with a composite key stores tuples with the same join attribute value in separate entries, using a concatenated index key of the primary key and join attribute. This setup enhances lookups with prefix filters, where the join attribute serves as the prefix. For example, as illustrated in Figure 2, n update involving the primary key $\{6\}$ and join attribute $\{E\}$ generates an index key $\{E6\}$, incurring an update cost of $O(L \cdot T \cdot \frac{e}{B})$. For point lookups, all LSM-tree levels must be examined since the entries with qualified join attributes may exist at multiple levels. In this case, it would involve at most L I/O times to access these levels. Additionally, given there are d matched entries, $d \cdot \frac{e}{B}$ I/Os are required to retrieve them. Hence the point lookup cost turns out to be $O(L + d \cdot \frac{e}{B})$. Although Composite Index generally offers robust update and lookup performance, its practicality may suffer due to increased space requirements and more compaction cost, particularly with more duplicated join attributes.

Another type of secondary index, the embedded index, is not included in our benchmark due to its prohibitive overhead for joins in LSM-based stores. In our attempt, it took several seconds to retrieve a single designated entry duo to its excessive CPU cost, making it impractical in our context.

Running example: For a data table with 10 million tuples with size of 64 bytes, a three-level LSM-tree should be constructed if the write buffer is 16 MB and the size ratio is 10. When the Bloom filter memory is set to 10 bits per key, as is presented in Table 3, the searching cost $(L' \cdot p)$ of Eager is neglectable, thus leading to a faster lookup performance compared to the other two types. Meanwhile, the update cost of Eager Index is much higher since the writing cost $(L \cdot T \cdot \frac{e}{B})$ is less than 1 while the additional cost $(L \cdot p + \lceil \frac{e}{B} \rceil)$ is greater than 1. Hence Eager Index benefits the query-intensive

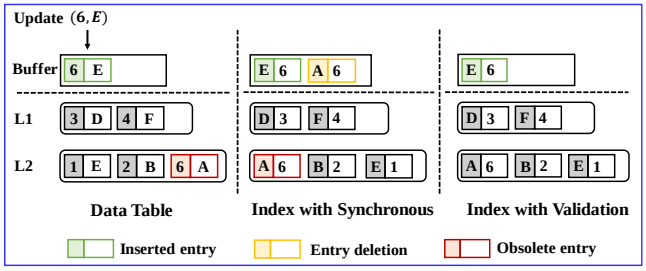

Figure 3: An illustration of updates with different consistency strategies. It is clear that Synchronous introduces additional update costs, while Validation cannot delete invalid entries effectively resulting in extra validating costs for query.

workloads like indexed nested loop join with high join frequency, which we will discuss in Section 4.

3.2 Consistency strategy analysis

Due to the out-of-place update mechanism, obsolete data entries in LSM-based systems may not be promptly removed, requiring consistent maintenance between data and index tables to ensure join result accuracy. This introduces challenges in LSM-based storage systems. Two main strategies, *Synchronous* and *Validation*, address these issues and differ in their update and lookup performances for different index types.

Algorithm 1: Update with Synchronous

Input: Data table R, index table R', update data (p_i, n_i)
Lookup p_i in R for the corresponding secondary key n_x
if n_x == $NULL$ **then**
  Insert (p_i, n_i) to R'
else
  Lookup n_x in R' for the corresponding primary key p_x
  Delete (n_x, p_x) in R'
  Insert (n_i, p_i) to R'
end

Algorithm 2: Query with Validation Strategy

Input: Data table R, index table R', queried secondary key n_i
Lookup n_i in R' for the corresponding primary key p_x
if p_x *is* $NULL$ **then**
  return $NULL$
else
  Lookup p_x in R for the corresponding secondary key n_x
  if n_x == n_i **then**
    return (n_i, p_i)
  else
    return $NULL$
  end
end

Synchronous. Synchronous immediately checks validity upon updates to maintain consistency. This strategy improves lookup performance but increases update costs due to additional synchronization as Figure 3 presents. To update the index LSM-tree, several steps are involved to verify the existence of associated data entry and insert index entry, which is described in Algorithm 1. Accordingly, the update and lookup costs for different secondary indexes are detailed in Table 3.

Validation. As Figure 3 present, for the Validation strategy, it simply inserts new entries to update the index LSM-tree without removing obsolete data. Therefore, the validity of entries should be checked during the join process to ensure the correctness of the results as is stated in Algorithm 2. Specifically, the joining cost of Validation includes both querying the index entry and verifying all associated primary keys, expressed as $O(d \cdot (L \cdot p + \lceil \frac{e}{B} \rceil)) + O(L' + d \cdot \frac{e'}{B})$ in Table 3, where d is the number of primary key duplicates per index entry. Validation therefore incurs lower costs during index building but higher costs during joining.

Covering Index. Besides the update method and consistency strategy in the index LSM-tree, index entries can be constructed as either covering or non-covering. A covering index includes the primary key and all tuple attributes, except the join attribute, within the index entry's value. Conversely, a non-covering index stores only the secondary and primary key. It results in smaller entry sizes which delivers improved update performance and degraded lookup performance. However, when all columns are selected in a query, it necessitates retrieving associated tuples from data LSM-tree through multiple point lookup operations, which incurs substantial additional costs.

Noted that the choice between covering and non-covering index entries is closely tied to the consistency strategy. Covering indexes complement Synchronous methods by enhancing lookup performance without querying data LSM-trees. In contrast, Validation pairs well with non-covering indexes, as both require data entry retrieval. While non-covering indexes with Synchronous can be effective in certain cases, they are not universally suitable for queries selecting columns beyond join attributes and primary keys.

3.3 Join algorithm analysis

In this section, we integrate the previously discussed index designs into three join algorithms: indexed nested loop join, sort-merge join, and hash-based join, in LSM-based key-value stores. A join method is the combination of specific join algorithm, secondary index type, and consistency strategy.

Indexed Nested Loop Join (INLJ). INLJ is a fundamental join that scans the left (outer) data table R and searches the right (inner) data table S for matching join attributes. This method's overhead includes scanning and searching costs. Scanning each entry in R's LSM-tree incurs a constant cost of $N_R \cdot \frac{e_R}{B}$. Without an index, searching requires scanning the entire right LSM-tree for each R entry, costing $N_R \cdot (N_S \cdot \frac{e_S}{B})$, which is unacceptable. However, if S is indexed, either by its primary key or a separate index LSM-tree, the cost is drastically reduced by the point lookup operation. As detailed in Section 3.1, point lookup costs depend on whether the search finds a match. Assuming a match ratio ϵ_R in R, the search cost is $(N_R \cdot \frac{e_R}{B}) + N_R((1 - \epsilon_R)Z_0 + \epsilon_R \cdot Z_1)$, where Z_0 and Z_1 represent the costs of empty and non-empty lookups shown in Table 3. The overall join costs for various scenarios are further detailed in Table 4.

Sort-Merge Join (SJ). SJ involves sorting tuples from the left and right data tables based on their join attributes to identify matched

Table 4: The theoretical analysis of the join cost and space complexity of three join algorithms under different scenarios. Please note that S and R represent the inner table and outer table for INLJ, respectively.

Method	Join algorithm	Data table S	Data table R	Space complexity	Join cost
NL-P	Indexed Nested Loop Join	Primary index	Regular column	$D(R)+D(S)$	$(N_R \cdot \frac{e_R}{B}) + N_R((1-\epsilon_R)Z_0(S) + \epsilon_R \cdot Z_1(S))$
NL-PS		Primary index	Secondary index	$D(R)+D(S)+D(R')$	$(N_{R'} \cdot \frac{e_{R'}}{B}) + N_R((1-\epsilon_R)Z_0(S) + \epsilon_R \cdot Z_1(S))$
NL-N		Regular column	Regular column	$D(R)+D(S)$	$(N_R \cdot \frac{e_R}{B}) + N_R(N_S \cdot \frac{e_S}{B})$
NL-NS		Secondary index	Regular column	$D(R)+D(S)+D(S)$	$(N_R \cdot \frac{e_R}{B}) + N_R((1-\epsilon_R)Z_0(S') + \epsilon_R \cdot Z_1(S'))$
NL-SS		Secondary index	Secondary index	$D(R)+D(S)+D(R')+D(S')$	$(N_{R'} \cdot \frac{e_{R'}}{B}) + N_R((1-\epsilon_R)Z_0(S') + \epsilon_R \cdot Z_1(S'))$
SJ-P	Sort-merge Join	Primary index	Regular column	$2D(R)+2D(S)$	$(N_S \cdot \frac{e_S}{B}) + 5(N_R \cdot \frac{e_R}{B})$
SJ-PS		Primary index	Secondary index	$D(R)+2D(S)+D(R')$	$(N_S \cdot \frac{e_S}{B}) + (N_{R'} \cdot \frac{e_{R'}}{B})$
SJ-N		Regular column	Regular column	$2D(R)+2D(S)$	$5(N_S \cdot \frac{e_S}{B}) + 5(N_R \cdot \frac{e_R}{B})$
SJ-NS		Secondary index	Regular column	$2D(R)+D(S)+D(S')$	$(N_{S'} \cdot \frac{e_{S'}}{B}) + 5(N_R \cdot \frac{e_R}{B})$
SJ-SS		Secondary index	Secondary index	$D(R)+D(S)+D(R')+D(S')$	$(N_{S'} \cdot \frac{e_{S'}}{B}) + (N_{R'} \cdot \frac{e_{R'}}{B})$
HJ-P	Hash Join	Primary index	Regular column	$2D(R)+2D(S)$	$3(N_S \cdot \frac{e_S}{B}) + 3(N_R \cdot \frac{e_R}{B})$
HJ-N		Regular column	Regular column	$2D(R)+2D(S)$	$3(N_S \cdot \frac{e_S}{B}) + 3(N_R \cdot \frac{e_R}{B})$

Table 5: The theoretical analysis of the multi-join cost of three join algorithms under different scenarios.

Method	Data table V	Data table W	Join cost	Multi-Join cost
NL-P	Primary index	Regular column	$(N_R \cdot \frac{e_R}{B}) + N_R((1-\epsilon_R)Z_0(S) + \epsilon_R \cdot Z_1(S))$	$L_V \cdot T \cdot \lceil N_V \cdot \frac{e_V}{B} \rceil + (N_V \cdot \frac{e_V}{B}) + N_V((1-\epsilon_V)Z_0(W) + \epsilon_V \cdot Z_1(W))$
NL-PS	Primary index	Secondary index	$(N_{R'} \cdot \frac{e_{R'}}{B}) + N_R((1-\epsilon_R)Z_0(S) + \epsilon_R \cdot Z_1(S))$	$L_V \cdot T \cdot \lceil N_V \cdot \frac{e_V}{B} \rceil + (N_W \cdot \frac{e_W}{B}) + N_W((1-\epsilon_W)Z_0(V) + \epsilon_W \cdot Z_1(V))$
SJ-P	Primary index	Regular column	$(N_S \cdot \frac{e_S}{B}) + 5(N_R \cdot \frac{e_R}{B})$	$L_V \cdot T \cdot \lceil N_V \cdot \frac{e_V}{B} \rceil + (N_V \cdot \frac{e_V}{B}) + 5(N_W \cdot \frac{e_W}{B})$
SJ-PS	Primary index	Secondary index	$(N_S \cdot \frac{e_S}{B}) + (N_{R'} \cdot \frac{e_{R'}}{B})$	$L_V \cdot T \cdot \lceil N_V \cdot \frac{e_V}{B} \rceil + (N_V \cdot \frac{e_V}{B}) + (N_{W'} \cdot \frac{e_{W'}}{B})$
HJ-N	Regular column	Regular column	$3(N_S \cdot \frac{e_S}{B}) + 3(N_R \cdot \frac{e_R}{B})$	$L_V \cdot T \cdot \lceil N_V \cdot \frac{e_V}{B} \rceil + 3(N_V \cdot \frac{e_V}{B}) + 3(N_W \cdot \frac{e_W}{B})$

pairs. Therefore, three primary phases should be entailed. First, all entries in the data LSM-tree are scanned into memory and sorted into several sorted runs, which are then written to disk. Due to memory constraints, it is impractical to sort all entries into a single run at once. Therefore, in the following phase, a k-way merge algorithm [8] is employed to consolidate these runs into a singular run, which is then saved back to disk as well. The final step reads this run in a streamlined manner to complete the join process. Hence the join cost sum up to be $5(N \cdot \frac{e}{B})$. When indexes are available, entries can be sequentially retrieved according to join attributes, eliminating the need for sorting and merging phases. As detailed in Table 4, SJ exhibits optimal join efficiency when both tables are indexed, a finding supported by our experimental data.

Hash Join (HJ). HJ constructs hash tables for the left and right tables using join attributes, ensuring tuples with matching attributes are grouped into the same buckets. During the join, corresponding buckets from both tables are retrieved to match tuples. For example, constructing a hash table for the left table R involves scanning all data entries, assigning them to buckets based on their hash values, and storing these buckets on disk. The buckets are then read during the join, incurring a total cost of $3(N_R \cdot \frac{e_R}{B})$. Notably, the efficiency of HJ is not affected by the presence of indexes. As indicated in Table 4, HJ generally incurs lower join overhead than SJ without indexes, but SJ becomes more advantageous as indexes are utilized. Further analysis of these algorithms' performance and characteristics is discussed in subsequent sections.

Runing example: Consider a join operation involving two data tables, each containing 10 million tuples with size of 64 bytes. Table S has a primary index, while table R has no index, and the ratio of matched entries to total entries in R (ϵ_R) is set at 1%. Additionally, the data block size is 4KB, and the Bloom filter uses 10 bits per key. Based on the results in Table 4, SJ and HJ exhibit comparable performance, while INLJ outperforms them due to efficient query operations (with small Z_0 and Z_1). When the entry size of table R (e_R) decreases from 64 bytes to 4 bytes, SJ's join cost decreases substantially until surpassing INLJ, whose performance remains stable due to the ceiling component in Z_1. Conversely, as e_R increases from 64 bytes to 2048 bytes, HJ can outperform SJ.

4 EVALUATING JOINS OVER LSM-TREES

We experimentally evaluate the influence of various factors on the performance of diverse join methods to summarize practical guidelines for choosing the most suitable methods under specific working conditions. To this end, we identify several important variants introduced in the theoretical analysis as presented in Tables 3 and 4, which can be classified into two categories: **join algorithm related factors** and **secondary index related factors**. We begin by examining the join algorithm related factors, including entry size, join frequency, and matching rate in Section 4.2 to Section 4.4 to develop foundational insights. Then analyze the impact of secondary index related factors, such as entry size, data distribution, and join frequency in Section 4.5 to Section 4.7 to refine our insights on determining the optimal combination of join algorithm

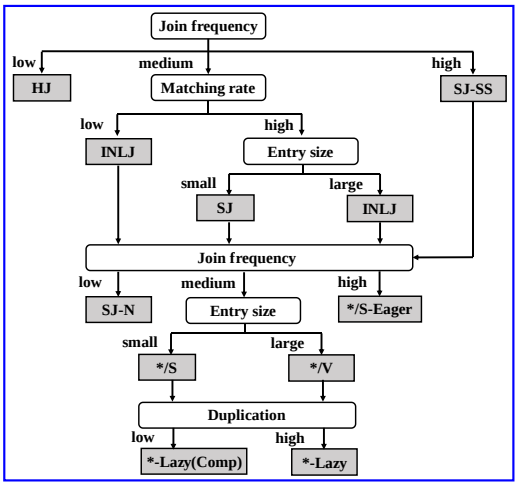

Figure 4: A sketch of a decision graph according to the theoretical analysis results in Table 3 and Table 4.

and secondary index. Additionally, we consider other factors that could impact the performance of join methods, such as the number of database updates and LSM-tree parameters, to gain a more comprehensive understanding in Section 4.8 and Section 4.9.

Hardware. All experiments are conducted on a server with an Intel Xeon Gold 6326 processors, 256GB DDR4 main memory, 1TB NVMe SSD, a default page size of 4 KB, and running 64-bit Ubuntu 20.04.4 LTS on an ext4 partition.

Implementation. We implement all 29 join methods within our proposed configuration space on a unified platform based on a prevalent LSM-based storage system RocksDB [24]. Since RocksDB does not inherently support joins, we tailored the workflows of various join algorithms and index designs specifically for this system. Additionally, many of these join methods have not been realized in existing LSM-based stores, which necessitates design from scratch. This required us to meticulously consider numerous implementation details. For instance, in Eager and Lazy index, some posting lists stored in the index entries can be extremely long, particularly with highly skewed or duplicated data distributions. This results in substantial overhead that significantly degrades system performance. To address this issue and enhance system efficiency, we employed practical C++ techniques for boosting string management of entries [57]. To construct the LSM-tree for data tables and index tables, we adhere to the default setting of RocksDB, where block size is set to 4096 bytes and the write buffer is set to 16MB. To emphasize the differences among various methods, the size ratio is set to 5 to include more levels, and the block cache size is set to 0.

We implement the sort-merge join following the streamlined approach with the total size of sorted segments set to 16MB. In hash join, we integrate the BKDR hash function [62] to the Grace Hash Join to reduce hash collisions with 16MB memory assigned to partition and probe the buckets.

We have compiled these join methods into an independent library that can be seamlessly integrated into RocksDB-based platforms with extensibility. In this library, we offer user-friendly APIs to assist users in instantiating desired join methods. These methods leverage RocksDB's internal structures for improved performance while maintaining compatibility across versions. Besides, we integrate RocksDB's transaction mechanisms to provide consistency guarantees for join operations. By eliminating additional dependencies, this library can be conveniently deployed on various platforms. Furthermore, since it is built upon native RocksDB, it can incorporate future updates to further enhance join efficiency.

Baselines. We examine all join methods in our configuration space each specified by particular join algorithm (i.e. INLJ, SJ, and HJ), secondary index type (i.e. Eager Index, Lazy Index, and Composite Index), and consistency strategy (i.e. Synchronous and Validation). We employ the notation presented in Table 3 and Table 4 to denote specific join methods concerning particular join algorithms with specific secondary indexes. For example, INLJ-NS/S-Eager refers to the indexed nested loop join applied to a scenario where the data table R is a regular column and S is indexed using an eager index with a synchronous consistency strategy.

Datasets. We incorporate four real datasets and two synthetic datasets in our experiments as listed in Table 6. These datasets encompass diverse typical workloads with distinct data distributions specified by duplication, matching rate, and skewness. Specifically, duplication refers to the average number of replicas of primary keys (c_r, c_s) or join attributes (d_r, d_s) in the data table. Aligned with the statements in Section 3.3, matching rate (ϵ) represents the ratio between the number of selected entries in a join over the total number of entries in a data table. We also examine the skewness (θ) of the data where larger values indicate more skewed distributions.

Following the existing works related to joins [48, 49, 65, 69, 72, 74], we examine three widely used benchmarks, Stack [51], IMDB [41], and SOSD [50] to derive our real datasets. For tailoring each benchmark to our task, we select two joinable attribute as the join attributes. *User* dataset selects the *question* table and *user* table from the Stack benchmark with *user_id* as the join attribute. This dataset involves high duplication and skewness along with a moderate matching rate. *Movie* dataset is extracted from IMDB benchmark including the *cast info* table and *movie info* table where *movie_id* column is selected for join. This dataset, similar to *User*, has high duplication and skewness while presenting a low matching rate. *Face* dataset utilizes *face* subset of SOSD benchmark and user IDs as join attributes without duplication and skewness. Please note that all tuples are involved in join operation, leading to the matching rate as 1. *Wiki*: dataset include *wiki* subset of SOSD benchmark. Distinct from *Face*, this dataset presents slightly skewed distribution with a few duplications [50].

Furthermore, we design two adjustable synthetic datasets which, respectively, follow distinct data distributions. Besides, we can flexibly vary their features (e.g., entry size, skewness, etc.) to generate diverse workloads thus significantly assisting us in exploring the

Table 6: Feature of Data Distribution in Benchmark Datasets.

Dataset		Duplication		Matching		Skewness	
		d_r	d_s	ϵ_r	ϵ_s	θ_r	θ_s
Real Database	*Movie*	4.1	5.6	0.72	0.99	0.89	0.85
	User	4.2	3.9	0.42	0.40	0.93	0.92
	Face	1.0	1.0	1	1	0	0
	Wiki	1.1	1.1	1	1	0.20	0.20
Synthetic Datasets	*Unif*	4.0	4.0	0.8	0.8	0	0
	Zipf	2.6	2.6	1	1	0.5	0.5

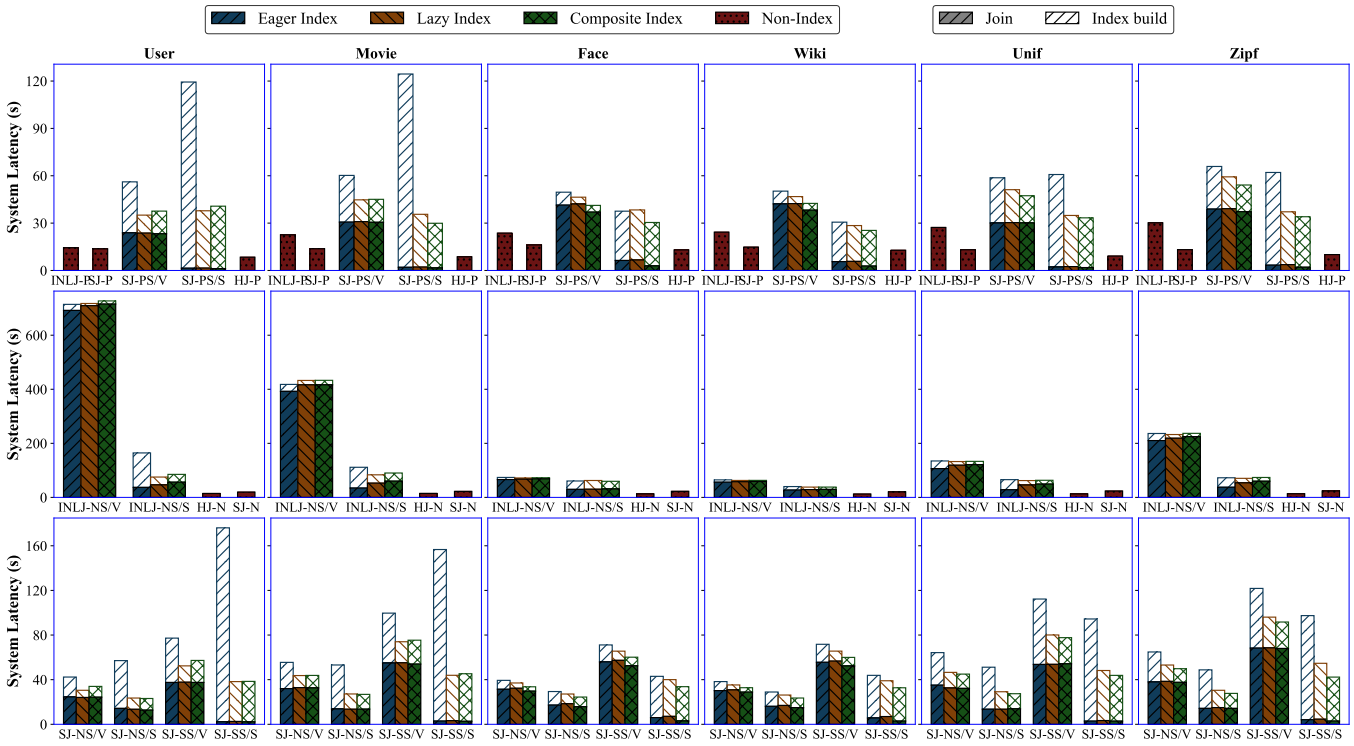

Figure 5: Overall experiments with default setting.

configuration space. *Unif* dataset incorporates a sequence of integers as the join attribute which follows the uniform distribution. Under default setting, it delivers high duplication and moderate marching rate. *Zipf* dataset adopts a widely utilized skewed distribution [30, 33, 44, 58, 66, 73], Zipfian, to generate the join attribute, which has moderate duplication and skewness by default.

The default setting of these datasets has been summarized in Table 6. We include 10 million entries in each dataset by default, and the entry size is set at 64 bytes, where both the primary key and join attribute occupy 10 bytes, resulting in a total dataset size of 0.5GB. This setup allows us to construct proper structures that effectively capture the key characteristics of LSM-trees while ensuring experimental efficiency. Figure 5 presents the performance of all join methods across all datasets under the default settings. Apart from the default setting, we further expand the scope of our evaluation by varying parameters mentioned above including entry size, entry number, and data distribution, with dataset size varing from 0.5GB to 40GB, to provide a more comprehensive analysis.

4.1 Current join methods in LSM-based stores can be suboptimal

Current LSM-based stores typically adhere to the traditional relational database convention of determining the join algorithm based primarily on selectivity. Additionally, they support specific secondary indexes based on common assumptions, leading to the neglect of many potentially effective index designs. Our experiments demonstrate that several factors significantly impact the relative performance and prioritization of different join methods as well, including entry size (Figure 6 (a)), join frequency (Figure 6 (b)), and matching rate (Figure 6 (c)). Furthermore, various index designs exhibit impressive performance under specific conditions, influenced by factors such as entry size (Figure 7), data distribution (Figure 8), and join frequency (Figure 10).

To the best of our knowledge, no existing works have thoroughly investigated how these factors influence the selection of join methods or carefully examined the underlying principles. Our study fills this critical gap by providing novel insights from extensive experiments to ease the selection of appropriate join methods.

4.2 Entry size vs. join algorithm selection

To examine the influence of entry size, we test different join algorithms on two datasets, *Movie* and *Face*, with entry sizes varying from 32 bytes to 4096 bytes. We evaluate a wide range of scenarios, including those without an index, a single index, and dual index, with the results presented in Figure 6 (a). In single index cases, we focus on configurations with a primary index to avoid the impact of index construction costs. For dual index scenarios, INLJ and HJ are excluded since, as demonstrated in Table 4, additional indexes do not lead to performance enhancement. Moreover, the results of configurations without an index typically present inferior performance, thus being placed in our technical report for space constraints. Furthermore, the secondary index Comp is employed by default, and the selection of secondary index type is further discussed in the subsequent subsections.

According to the results, SJ and HJ deliver comparable join latencies in different cases for both datasets. This aligns with our

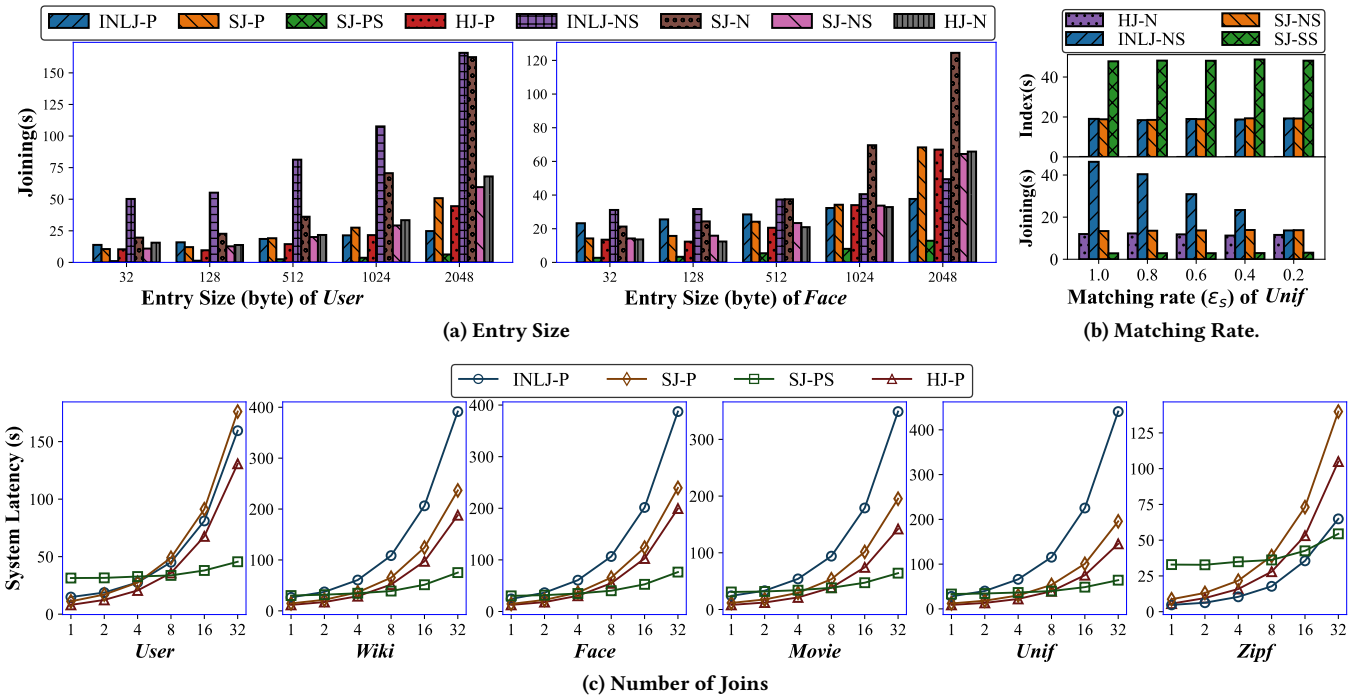

Figure 6: The performance of different join algorithms varies significantly with changes in entry size, join frequency, and matching rate, thus impacting their optimality. (In (a), SJ-PS/S-Comp requires additional index building latency (seconds) from left to right as follows: 29, 43, 278, 627, 1162 (*User*); 25, 43, 239, 597, 1149 (*Face*).)

theoretical analysis, indicating that the join costs of SJ-P and HJ-P are $(N_S \cdot \frac{e_S}{B}) + 5(N_R \cdot \frac{e_R}{B})$ and $3(N_S \cdot \frac{e_S}{B}) + 3(N_R \cdot \frac{e_R}{B})$, respectively, which are roughly identical when the left data table R and right data table S have the same data scale and entry size. Additionally, this indicates that the join cost of these two join algorithms increases linearly with the entry size, which is also consistent with the experimental results. In contrast, the performance of INLJ is more robust to increases in entry size. As the entry size varies from 32 bytes to 4096 bytes, the join latency of HJ scales up by 1600% in the *Movie* dataset. While the join latency of INLJ only increases by 75%. According to the analysis in Section 3.3, the overhead of INLJ joins consists of scanning costs and searching costs. The scanning cost increases with entry size, while the searching cost is determined by the lookup cost of the associated index, which remains constant as long as the entry size is smaller than the block size. Therefore, although INLJ performs worse with small entry sizes, it gradually outperforms SJ and HJ as the entry size grows, as observed in both the *Movie* and *Face* datasets. To be specific, the join latency of INLJ is approximately 13 seconds higher than that of SJ and HJ in the *Movie* dataset when the entry size is 32 bytes. Conversely, INLJ outperforms both HJ and SJ when the entry size exceeds 512 bytes.

Notably, sort-merge join with dual index (SJ-PS) exhibits impressive join efficiency in Figure 6 (a), due to the efficient range scan of LSM-tree. However, it requires an additional index which incurs significant index building overhead, thus degrading the overall performance. Meanwhile, the impact of index building cost varies with join frequency in a workload, leading to different selections of join algorithms, which we explore in the next subsection.

We also analyze the performance in the scenarios when the join attributes are non-primary columns in both tables. As shown in Figure 6 (a), INLJ methods approach or even outperform HJ and SJ methods as the entry size increases.

4.3 Join frequency vs. join algorithm selection

To explore the impact of join frequency on the selection of join algorithms, we vary join frequencies from 1 to 32 and evaluate the performance of various join algorithms using two real datasets, *User* and *Wiki*. Here, join frequency refers to the number of join operations in a given workload. For instance, if the join frequency is set to 2, the dataset containing 10 million updates is divided into two subsets of 5 million updates each, with a join operation applied after processing each subset. As a result, the total number of updates remains constant among all experiments, regardless of changes in join frequency. We assessed the system latency including joining latency and index building latency, as shown in Figure 6 (b). The results in both datasets demonstrate that SJ-P/S-Comp performs poorly when join frequency is low but gradually outperforms all other algorithms as join frequency increases. For example, with a join frequency of 1, the system latency of SJ-PS is nearly five times that of the best-performing algorithm, hash join with primary index (HJ-P). However, when the join frequency increases to 32, SJ-P/S-Comp turns out to be the best algorithm with system latency 50% of HJ-P. As indicated in Table 4 and Figure 6

(a), SJ-PS/S-Comp achieves extremely high join efficiency since the indexes of the two data tables enable sequential retrieval of tuples according to the join attribute without a sorting process. However, it comes with the expense of additional overhead on index construction. With higher join frequencies involved, the joining costs for SJ-P, INLJ-P, and HJ-P increase significantly. In contrast, the joining latency of SJ-P/S-Comp increases moderately, with the index building latency remaining constant. Consequently, SJ-P/S-Comp is presenting increasingly attractive performance as higher join frequency is considered.

Moreover, INLJ-P typically has higher system latency than SJ-P and HJ-P when entry size is small, as discussed in Section 4.2. However, INLJ outperforms SJ in the *User* dataset which contradicts our current sense. Comparing these two datasets, *User* presents distinct matching rates and data distribution (i.e. duplication and skewness), suggesting the importance of these influential factors on join method selection, which are analyzed in the following.

4.4 Matching rate vs. join algorithm selection

As mentioned in Section 4.3, the matching rate can affect the performance of join algorithms in certain databases. Hence it is important to understand how this influence affects each join algorithm and, consequently, impacts the selection of the most appropriate join algorithm. To this end, we evaluated the performance of different join algorithms on the *User* dataset, with matching rates decreasing from 1.0 to 0.2, under the scenario where the join attribute is a non-primary key in both data tables R and S. For a more detailed analysis, we decomposed the system latency into joining latency and index building latency, as shown in Figure 6 (c). It is evident that the join cost of INLJ is positively correlated with the matching rate. As the matching rate decreases, there are fewer matched entries in the data table, leading to an increase in the number of empty point lookup operations during the join process. According to Table 3, empty point lookups incur lower costs than non-empty point lookups, causing the join cost of INLJ to decrease accordingly. In contrast, the joining latency of HJ and SJ is not affected by the matching rate since all entries in the data table should be extracted regardless of whether they match the join condition or not.

Moreover, index building latency is unaffected by matching rate as index updates do not differentiate matched entries. Specifically, HJ does not involve a secondary index, thus incurring no additional cost. INLJ-NS and SJ-NS construct a secondary index for the data table S, resulting in similar index building costs, while SJ-SS incurs a much higher overhead as secondary indexes are built for both data tables. Therefore, INLJ is preferred when a low matching rate is involved. Moreover, the experiments and analysis indicate that the matching rate only affects the performance of join algorithm related to lookup operations.

This experiment also reveals the general impact of secondary indexes on the performance of different join algorithms. The update performance of secondary indexes affects the index building latency of INLJ and SJ, while the lookup performance mainly influences the joining latency of INLJ. More complete and in-depth analysis of these influences is provided in the following subsections.

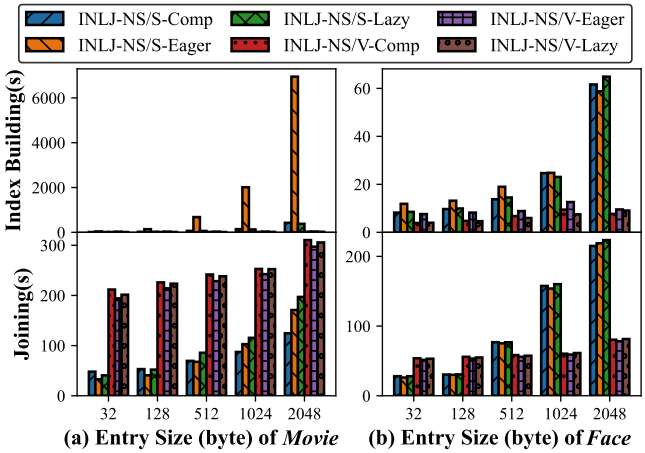

Figure 7: The Impact of entry size on index designs.

Remark: After examining the performance of join algorithms across various influential factors, we achieve the conclusion that **the selection of join algorithms is not simply subject to selectivity**. An increasing entry size or decreasing matching rate favors INLJ. Meanwhile, a higher join frequency dilutes the significance of index building overhead, making join algorithms that involve indexes, such as INLJ and SJ, more favorable. In cases where index building cost can be neglected, SJ with two indexes is more favorable.

4.5 Entry size vs. Index design

To examine the impact of entry size on index design, we evaluated different entry sizes on *Movie* and *Face* datasets, which are more skewed and more uniform (less duplicated), respectively. As shown in Figure 7, a notable observation is that for Eager Index with Synchronous on skewed distribution, the index building cost is extremely high. For instance, the index building cost approaches 7000 seconds when entry size grows to 2048 bytes for *Movie*. This can be attributed to Eager Index exhaustively merging all entries with the same keys together. Moreover, in skewed datasets, some entries occur more frequently, leading to dramatically larger entry sizes. Synchronous has to retrieve these large entries repeatedly to remove stale values. In this process, large entry sizes can lead to a more serious burden. When entries are smaller, the check cost in Synchronous ($O(L \cdot p + \lceil \frac{e}{B} \rceil)$ in Table 3) is expected to be bound to one I/O. When entry size grows larger, the sizes of more merged entries are expected to exceed one I/O. This explains why large entry sizes can lead to a more serious burden for Eager Index with Synchronous. While Composite Index and Lazy Index do not eagerly merge the entries, their entry sizes are more uniform and relatively small compared to Eager Index, so their index building costs do not grow as dramatically.

For uniform datasets such as Face, the index building costs of Synchronous do not increase as dramatically, since their entry sizes remain relatively stable without many merges. However, the join costs of Synchronous exceed those of Validation. This is because the dataset is less duplicated, so Validation does not need to check

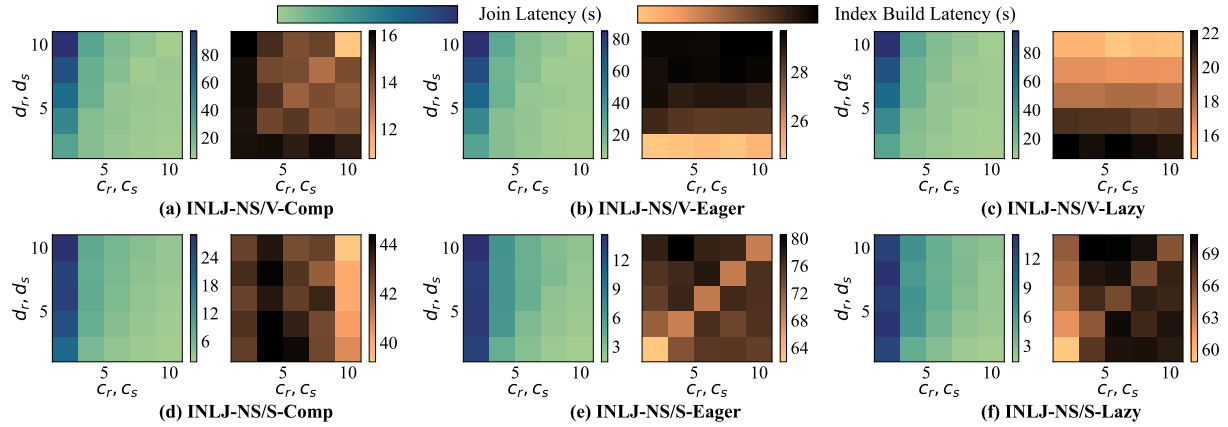

Figure 8: The performance of various index designs with diverse primary key and join attribute duplication in *Unif* dataset.

the validity for each entry repeatedly in INLJ. Moreover, Synchronous with covering Index has to maintain a large index table. As analyzed in Table 3, the point lookup costs of all indexes are related to $O(L)$, which increases with entry size ($L = \log_T \frac{N \cdot e}{M}$). Validation only keeps the primary keys and join attributes, which results in stable index table sizes. This is why Synchronous costs more than Validation in joining when entry size is large. In summary, for uniform and less duplicated data distributions, Validation maintains stable performance, thus being preferable when entries are large. For skewed data, the eager index is not recommended due to its significant index building time, especially for large entry sizes. The composite key index is more favorite given it generally delivers sound performance. However, this preference is also subject to the join frequency as we introduce in Section 4.7.

4.6 Data distribution vs. Index design

We examine data distribution's impact on index design in two scenarios. First, we analyze the duplication in uniform datasets by varying the duplicates of primary key (c_r, c_s) and join attribute in a grid pattern on *Unif* dataset. Second, we investigate the impact of join attribute skewness (θ_r, θ_s) on *Zipf* dataset.

Duplication. Figure 8 reveals several key insights. Most notably, as primary key duplicates increase, join latency decreases. This is attributed to LSM-tree merging identical primary keys, effectively reducing the data scale. For all Validation methods (Figures 8(a)-(c)), join costs increase with higher join attribute duplication. This aligns with our analysis in Table 3, as it reflects the increased point lookup cost for Validation. For instance, in the case of Eager Index with Validation, the cost is represented as $O(d \cdot (L \cdot p + \lceil \frac{e}{B} \rceil)) + O(L' \cdot p + \lceil \frac{e'}{B} \rceil)$. As the duplication d increases, so does the overall cost. In the Synchronous approach, only Composite Index shows a slight increase in join latency with higher join attribute duplication (Figure 8(d)). This is because Eager Index and Lazy Index can merge more identical keys into single entries, consistent with our analysis.

Regarding index building, Eager Index demonstrates increased latency with higher join attribute duplication (Figure 8(b)), while Lazy Index shows an opposite trend (Figure 8(c)). This divergence occurs because Eager Index must perform non-empty point lookups for each duplicated join attribute, leading to higher index building latency. Conversely, Lazy Index merges identical join attributes through the LSM-tree's natural compaction process, which not only avoids extra costs but also reduces the index table size. For Synchronous methods, the trend is less clear-cut. Higher primary key duplication implies that Synchronous must check more entries, but it also results in a smaller index table scale. These opposing factors can either increase or decrease index building costs. Consequently, for Composite Index (Figure 8(d)) and Lazy Index (Figure 8(f)), index build costs remain relatively stable.

Skewness. Figure 9 reveals striking patterns in index building and join latency as data skewness increases. Notably, Eager Index demonstrates exceptionally high index building latency, while all Validation methods exhibit dramatically high join latency. When skewness reaches 0.7, the eager index enhanced INLJ with synchronous (INLJ-NS/S-Eager) and validation (INLJ-NS/V-Eager) strategy require 767 and 172 seconds for index building, respectively, while other methods take less than 25 seconds. This substantial difference can be attributed to the increased frequency of validity checks for high-occurrence entries as skewness grows. Furthermore, Eager Index's practice of merging identical keys results in potentially very large entry sizes for high-occurrence items. This issue is exacerbated in the Synchronous approach, which must frequently retrieve these large entries to remove stale primary keys from the value.

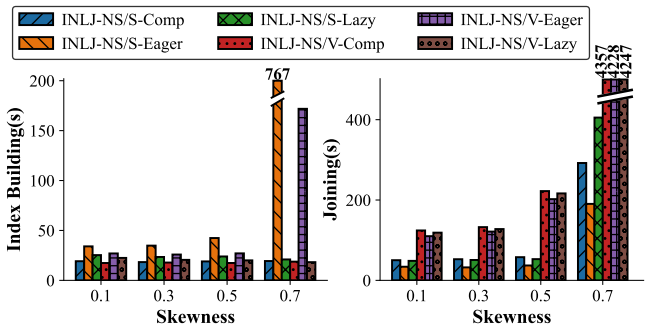

Figure 9: The performance of various index designs with varying skewness in *Zipf* dataset.

In contrast, Lazy Index and Composite Index, which don't eagerly merge entries, maintain more manageable entry sizes.

On the join latency front, Validation methods incur significantly higher costs. As skewness approaches 0.7, all Validation methods require over 4000 seconds to complete the join operation, while other methods take only a few hundred seconds. This substantial difference can be partially explained by our analysis in Table 3, where each Validation method includes a lookup term of $O(d \cdot (L \cdot p + \lceil \frac{e}{B} \rceil))$. In our experimental setup, as the skewness of both tables increases, this term is effectively multiplied by another factor of d in INLJ, leading to the observed high join latencies. In summary, Lazy Index and Composite Index with Synchronous approach demonstrate superior resilience to data skewness.

4.7 Join frequency vs. Index design

We examine the impact of join frequency on index design by varying the join frequency on two datasets: *User* and *Wiki*. As illustrated in Figure 10, the overall costs of Validation methods grow more rapidly than those of Synchronous methods, regardless of whether the dataset is uniform (*Wiki*) or skewed (*User*). For instance, on the *User* dataset, the costs of all Validation methods increase from 460-500 seconds to around 4800-5100 seconds as join frequency increases. In contrast, the costs of all Synchronous methods consistently remain below 630 seconds. This significant difference in performance can be attributed to the fundamental design of these methods. Validation methods do not maintain validity information. Consequently, for each join operation, they must check the data table for every matching entry. This results in repeated validity checks as join frequency increases. Synchronous methods, on the other hand, verify the validity of each entry only once during the index building phase. This approach saves $O(d \cdot (L \cdot p + \lceil \frac{e}{B} \rceil))$ I/Os compared to Validation for each lookup in subsequent joins.

Moreover, Eager Index demonstrates the best overall performance among the three types of indexes. For instance, on the *User* dataset, Eager Index with Synchronous completes 32 joins in 390 seconds, while Composite and Lazy Index require 630 and 570 seconds. Eager Index saves 38% and 32% overall latency respectively, which will become more pronounced as join frequency increases. This advantage stems from Eager Index's lower join cost, which can be guaranteed as one entry lookup ($O(L' \cdot p + \lceil \frac{e'}{B} \rceil)$ as shown in Table 3). In contrast, the other two index types may scatter entries

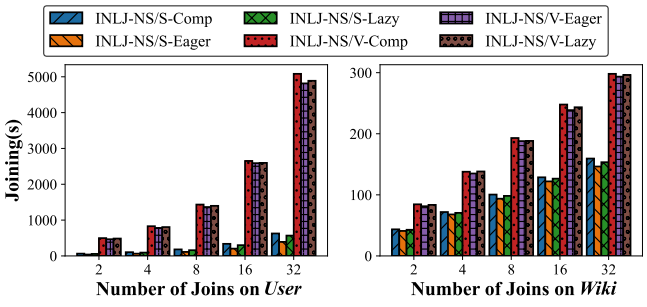

Figure 10: Different index designs deliver varied performance facing increasing join frequency.

with the same keys across different levels of the LSM-tree, necessitating multiple retrievals for a single matching data entry. The performance gap seems relatively modest with Validation methods as their costs are dominated by the check process of Validation. In summary, Synchronous and Eager Index methods demonstrate superior performance as join operations become more frequent.

Remark: After examining the performance of indexes and consistency methods across various influential factors, we can confidently conclude that **the absence of Eager Index and Validation in conventional LSM-tree databases is a missed opportunity**. Eager Index outperforms the other two indexes when joins are more frequent (e.g., more than twice per 10 million updates in our setting). Moreover, Validation outperforms Synchronous when entry sizes are large (e.g., exceeding 512 bytes on the *Face* dataset).

4.8 Impact of increasing database updates

The number of database updates influences the levels and entries, affecting multiple components of system overhead, such as join cost and index lookup cost. This relationship is complex and challenging to evaluate through theoretical analysis alone. Therefore, we varied the number of database updates to measure the cumulative join latency and cumulative index building latency of different join algorithms, secondary index types, and consistency strategies. Specifically, we divided the dataset into equal-sized subsets and performed a join operation after processing each subset. Upon completion of each join operation, we record the joining latency and index building latency that are then added to the previous measurements to obtain the cumulative results. We conducted experiments on two datasets, *Unif* and *Zipf*, with associated results presented in Figure 11.

In uniform datasets, the cumulative join latency for all methods has a quadratic growth, indicating that as the data scale expands, the joining cost for all methods increases linearly. This observation is supported by Table 4, which shows that almost every component is linearly related to the number of entries. For skewed datasets, as the duplicates of join attributes increases, the joining cost for INLJ methods exceeds that of SJ, and the cumulative latency trend increases more rapidly compared to uniform datasets. This escalation is attributed to the exponential growth in the number of duplicate join attributes in skewed data, which necessitates significantly more point lookups. However, when the skewed join attribute also serves as the primary key, this phenomenon is less noticeable because the LSM-tree can merge many duplicated keys, thus reducing the data scale and the number of required point lookups.

In terms of index building, the cumulative latency increases linearly in both datasets, indicating that the cost of incrementally updating the index with each partition of uniform data remains consistent. As analysed in Table 3, the number of levels is almost the sole dynamic factor in each term, and these levels do not change rapidly during index construction. So the expected cost of index building remains relatively stable as the data volume increases. Furthermore, the index building cost for the Eager Index also climbs more steeply. This is because the frequently occurring join attributes produce enlarged index entries, which requires more resources to access.

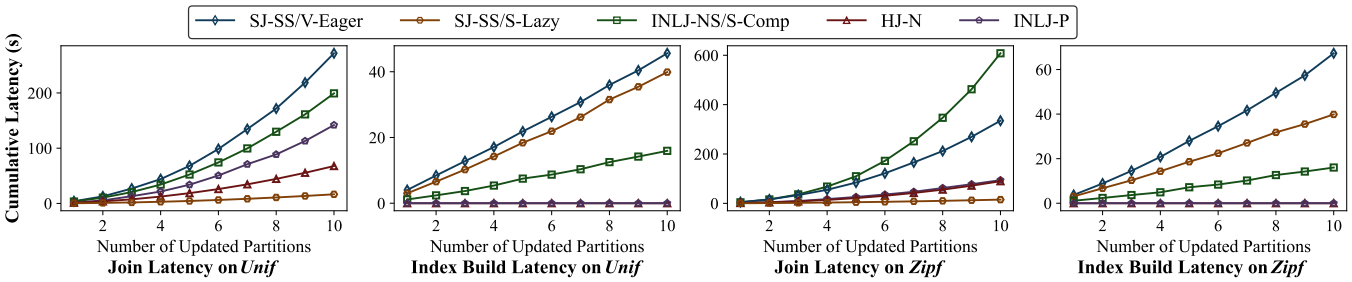

Figure 11: The performance of different join methods under increasing database updates.

Remark: In uniform datasets, join and index building costs grow steadily across methods, **thus our selection criteria is still applicable**. For highly skewed datasets, we recommend HJ or SJ with Composite or Lazy Index to deal with large-scale updates.

4.9 Impact of Tuning LSM-tree

To get more specific insights of the interaction of LSM-tree and joins, we dive into the specific parameters of LSM-tree, including size ratio, buffer size, cache size and bits per key for Bloom filters.

Size ratio. Our examination begins with the size ratio, a crucial factor in compaction policies, within the standard leveling compaction approach utilized by RocksDB. When we escalate the size ratio from 2 to 100 as shown in Figure 12, there's a noticeable decrease in the joining latency for INLJ, such as dropping from 60 seconds to 30 seconds when a composite index is incorporated with synchronous strategy (INLJ-NS/S-Comp). This reduction can be attributed to the decrease in the number of LSM-tree levels as the size ratio increases. Consequently, for Lazy Index, there are fewer versions of secondary keys, and for Composite Keys, the scatter of secondary keys across levels diminishes. This leads to a reduced need for *Seek* operations during lookups, enhancing efficiency. On the other hand, Eager Index maintains a singular version of secondary keys, rendering it unaffected by variations in the number of levels, and hence, its joining latency remains stable irrespective of the size ratio settings. For SJ and HJ, since they do not engage in any lookup operations, changes in the size ratio have no impact on their joining latency.

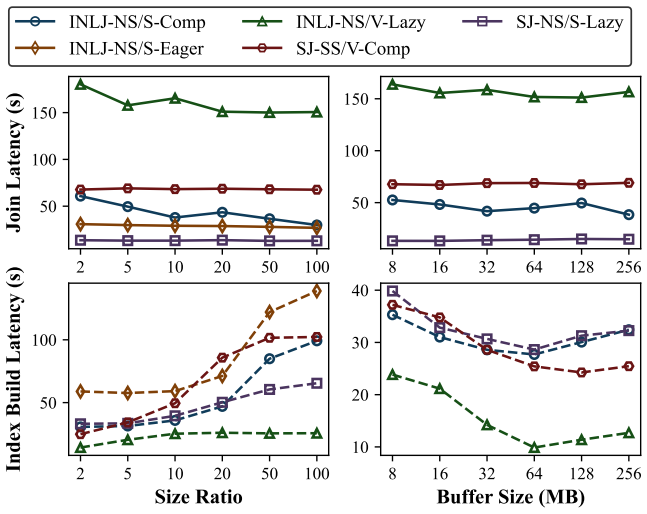

Figure 12: The impact of size ratio and write buffer.

Regarding the index building latency, the trend displays a general increase. For instance, if two composite indexes are built for sort-merge join with validation strategy (SJ-SS/V-Comp), the index building time grows from 25 to 105 seconds as the size ratio increases from 20 to 100. The reason behind the increase is that the increment of size ratio forces the entries to be involved in more large compaction which incurs higher update cast as Table 3 presents. Therefore, for INLJ, a relatively larger size ratio is beneficial when joins are frequent, while a smaller size ratio is preferable when joins are less frequent. For SJ and HJ, employing a smaller size ratio during index building is advisable as it has minimal impact on their joining costs.

Write buffer. Memory allocation has garnered significant interest within the LSM-tree community [31, 46, 59]. Our investigation concentrates on examining the memory allocated to the writer buffer, Bloom filters, and block cache. First, we examine the write buffer. Within RocksDB's default compaction policy, adjustments to the write buffer size affect join latency for INLJ methods with Composite and Lazy Index. This impact mirrors the effects observed with changes in size ratio, primarily by reducing the number of levels and facilitating the merging of entries in index tables. However, for Validation, this effect is less pronounced as its joining cost is predominantly influenced by the validation process, as discussed previously. For SJ and HJ, as they do not rely on point lookups, changes in the number of levels do not affect their performance.

For index building latency, theoretically, increasing the write buffer size should reduce latency. However, this is not consistently observed in practice. As illustrated in Figure 12, when the write buffer is increased to 256MB, the joining costs exceed those observed with a 64MB buffer. This discrepancy is due to larger write buffers causing more substantial flush operations, resulting in spikes in disk I/O that are less efficient than a uniform compaction schedule. In summary, we recommend a moderately large write buffer size to optimize both joining and index building processes.

Bloom filter. Next, we explore Bloom filters, which allocate memory for each key based on a specified number of bits. Increasing the bits per key reduces the false positive rate, thereby decreasing lookup costs. As shown in Figure 13, enhancing the bits per key effectively reduces the joining latency for various configurations of INLJ. For example, the joining latency for INLJ-NS/S-Eager can decrease from 40 to 28 seconds. This improvement is primarily due

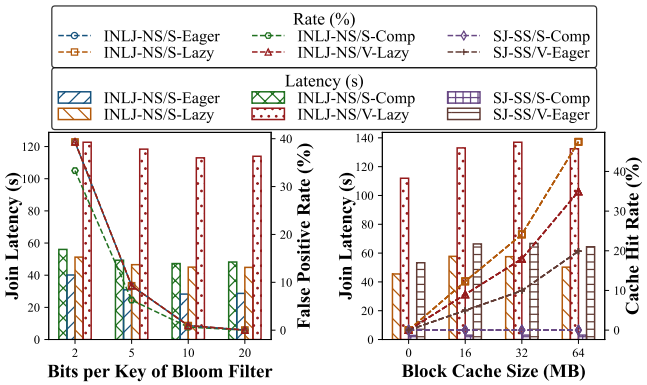

Figure 13: The impact of block cache size and Bloom filter.

to the significant reduction in the false positive rate, which drops from 40% to less than 1%. It is important to note that the rate of decrease in false positives begins to taper off after reaching 10 bits per key, consistent with theoretical predictions [18, 19]. Therefore, we recommend a moderately large number of bits per key for Bloom filters (e.g. 10) to balance efficiency in INLJ operations and memory usage.

Block cache. Lastly, we explore the role of the block cache in RocksDB. Theoretically, increasing the memory allocated to the block cache should decrease the joining latency for methods like INLJ and Validation. However, as depicted in Figure 13, the join latency initially increases and then decreases as the block cache size expands. This pattern occurs because when the cache size is small, replacements happen more frequently, requiring more CPU effort to manage and retrieve data. In contrast, SJ without Validation and HJ are unaffected by the block cache size. These methods scan all data in the LSM-tree once, negating the benefits of caching. In summary, we recommend relying on either the default page cache provided by the system or using a large block cache for methods like INLJ and Validation to optimize performance.

Remark: Larger size ratios enhance INLJ's joining but increase index building costs. Writer buffer faces similar trade-offs. So we recommend moderate size ratios (around 10) and writer buffers (64-128MB). For INLJ's lookup efficiency, 10 bits per key in Bloom filters suffice. Lastly, block cache can be omitted if adequate page cache is available.

5 DISCUSSION

Here it is time to answer the question in title – *are joins over LSM-trees ready?* The experiments and analysis in this benchmark indicate that there is still considerable potential for optimizing join method selection. The join methods used by existing databases cover only a small portion of our proposed configuration space and often fail to achieve optimal performance. Additionally, the efficiency of different joins varies significantly under many conditions. For instance, in Figure 6 (b), SJ-PS/S-CI could achieve more than 8x better latency than INLJ-P. Fortunately, we have derived valuable insights that enhance our understanding of this topic and offer practical takeaways for developers in selecting appropriate join methods. Moreover, we point out some constraints of our benchmark and suggest future directions to guide development.

5.1 Insights and takeaways

One Method Does Not Fit All. Figure 1 summarizes the selection strategies for different join components, demonstrating that each method could excel under specific conditions, with no single method dominating universally. For instance, HJ-P shows competitive performance in many scenarios as illustrated in Figure 6. However, it is outperformed by SJ-PS with frequent joins and by INLJ-P when dealing with large entry sizes. This also suggests an important takeaway message for joins over LSM-tree: ***Selection of join algorithms is influenced by multiple parameters (e.g., entry size, join frequency, and data distribution), instead of only subject to selectivity.***

Parameters matter unequally in join method selection. Among the parameters examined, data distribution, join frequency, and entry size are the most influential ones. Specifically, ***Data distribution and join frequency matter in determining secondary index types.*** Eager is preferred in workloads with frequent joins. When highly duplicated or skewed join attributes are involved, Eager excels in join performance, while Lazy and Comp offer lower index-building costs. ***Choose consistency strategies considering the entry size and join frequency.*** Synchronous with covering index excels in frequent joins with small entries, otherwise, Validation with non-covering index is more efficient. For skewed data, avoid combining Synchronous with Eager, or Validation with INLJ, due to the extensive additional overhead. In comparison, some parameters only affect specific components (e.g., Bloom filter only adjusts the performance of INLJ), or simply influence system performance without apparently altering the priority of join methods, such as database updates, size ratio, buffer size, and block cache size. Identifying these key parameters would greatly simplify the join method selection.

Secondary indexes are not always necessary. Secondary indexes enhance the join efficiency and robustness of INLJ with Bloom filters, as well as reduce sorting costs via the LSM-tree index. However, we should not overlook the additional index build cost they incur, as this cost can be substantial, sometimes outweighing the improvement in join. Therefore, secondary indexes may not always have a positive impact, particularly in cases with large entry sizes, high skewness, and infrequent joins.

Please note that we focus on the binary join over LSM-tree in our analysis, while the derived insights are still useful for multi-way joins. The topic of multi-way joins has been extensively studied in relational database systems, and our work can be seamlessly integrated as a component of these solutions. When the decomposition approach is involved [38, 52–55], our insights can be applied to optimize each binary join. For example, Bao [39] could use our cost model to reevaluate the cost of each step to generate more efficient query plan trees. In other scenarios where specialized techniques are used [39], our analysis of join costs can be incorporated to model specific operations, such as modeling the hash value computation cost in hash trie joins [39]. Furthermore, it is also interesting to benchmark specific multi-way join solutions to derive more detailed guidelines on method selection in future work.

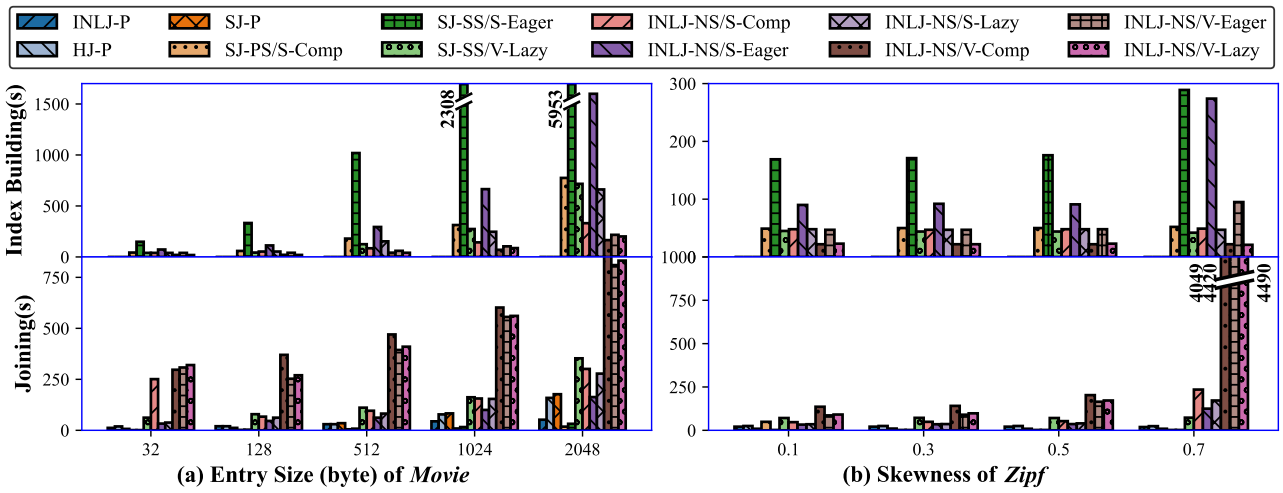

Figure 14: Extending to other LSM-based systems: a case study on Pebble.

5.2 Future Directions

LSM-based stores with different implementations and tuning parameters may present varied join performance thus yielding distinct insights, considering our takeaways are based on experiments with RocksDB. Meanwhile, we can easily extend our study to these scenarios by adjusting the configuration space by including associate parameters and reevaluating the cost model to derive new insights. For example, platforms utilizing a key-value separation strategy like Wisckey [17, 44] achieve higher update efficiency and may benefit from SJ due to the quick index construction. In such cases, the selection strategy should be reevaluated. However, our proposed configuration space can still provide effective guidance in designing new evaluation frameworks. Moreover, our insights and some takeaways may still be applicable since they reveal the fundamental principles of join operations over LSM-trees. Besides, we also suggest some promising directions for future works to enhance joins over LSM-trees. One area of focus is optimizing join algorithms specifically for LSM-trees and modern hardware [71] to further boost performance. Another potential improvement lies in incorporating novel LSM-style [25] or hybrid secondary indexes [40] to enhance overall performance and more flexibile update and query trade-off. Moreover, exploring advanced consistency strategies to maintain data integrity while reducing the query or update overhead is also crucial. As data scale increases, distributed strategies may be important to address the challenges of handling large-scale datasets and higher transaction volumes. Additionally, platform-specific optimizations, such as re-optimization schemes [40] and normalization approaches [16], could be further analyzed to identify the preferences for particular join methods.

5.3 Extending to other LSM-based Systems

To validate the generality of the insights presented in this paper, we conducted additional experiments on another LSM-based system to evaluate the performance of each join method. Among these systems, Pebble [16], the core engine of CockroachDB [70] written in Go, has garnered significant attention in the industry. Hence we select this storage to conduct a case study which results are presented in Figure **??** (a). As expected, although the exact performance on Pebble differ from those on RocksDB, the relative differences in the costs of different join methods, as well as the trends in performance changes across different settings, are apparently consistent with the results observed on RocksDB. Specifically, the joining latency for the HJ and SJ methods increases almost proportionally with the entry size, whereas the INLJ methods exhibit a slower increase. For instance, when selectivity is moderate, INLJ-P surpasses HJ-P and SJ-P in joining efficiency when the entry size reaches 512B. These observations are consistent with the phenomena we observed in Figure 6a. Additionally, in Figure **??** (b), the joining latency for the INLJ methods and Validation increases significantly under higher skewness, due to more frequent retrieval of higher-frequency data. When skewness reaches 0.7, the joining latency for all INLJ methods with Validation exceeds 4000 seconds, whereas other methods remain below 200 seconds. At the same time, the index building times for both Eager Index and Synchronize are noticeably longer than those of other methods. These results are also similar to the findings from the RocksDB experiments in Figure 9. Therefore, despite differences in implementation, since our insights are based on the theoretical cost model of LSM-tree, they can be generalized to other LSM-based systems.

For the systems encompassing a subset of our configuration space, we can evaluate the associated space and utilize the corresponding insights or guidelines. In cases where join performance is altered by hybrid data structures or specific optimizations on certain join methods, we can adjust our findings to them with reevaluation since the instinct of each join method remains consistent.

5.4 Extend to other Hash Joins

As we mentioned, we selected the most representative join methods to analyze their performance. To further validate the generality of the insights presented in this paper across more specialized join methods, we also included hybrid hash join [23, 67] as a case study.

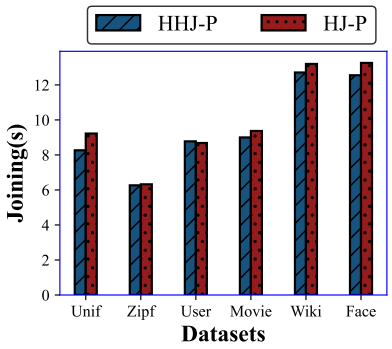

Figure 15: An illustration on the performance of grace hash join and hybrid hash join.

Compared to the default grace hash join used in this paper, hybrid hash join maintains an in-memory hash table for a portion of the table, potentially reducing I/O costs slightly. As shown in Figure **??**, across all datasets, hybrid hash join consistently exhibits a similar join cost to grace hash join, with a difference of only around one second. This is because, in disk-based joins, the amount of data typically far exceeds the memory budget, meaning hybrid hash join can only store a limited portion of data in memory, offering only marginal optimization.

6 RELATED WORK

LSM-tree stores. Extensive research has been conducted on LSM-trees, typically employing theoretical analysis to optimize parameters within a given design space, such as size ratio, compaction policy, and Bloom filters [18–21, 31, 32, 43, 47, 58]. Additionally, works focusing on self-designing data structures, such as Cosine [14], Design Continuums [34], and Data Calculator [35], have been proposed to address specific challenges including varying data distribution, concurrency, and evolving hardware. We agree that these methods could potentially improve join performance, while they typically enhance system performance by including more data structures like B+ tree in their configuration space instead of focusing on optimizing a certain type. Meanwhile, our results can also be utilized in these works to analyze the LSM component and optimize toward join operations since our insights are general for LSM-tree structure.

Moreover, there are many LSM-based industrial data stores [4, 5, 11, 13, 24, 27, 70] that have supported join operation. While they typically provide several join methods for users to select by themselves. Thus, a comprehensive benchmark study of joins over LSM-trees is still lacking. Some of the closest works focus on secondary indexes and consistency strategies [45, 64, 71]. Our study aims to address this gap.

Disk-based joins. Disk-based joins differ from in-memory joins [30, 65, 66] primarily due to their larger data volumes, necessitating more disk I/O operations. While extensively studied in relational databases [26, 37, 41, 42, 60, 63], traditional wisdom suggests that join costs mainly depend on selectivity or cardinality [3, 28, 69, 72, 75]. However, the LSM-tree context presents a different scenario. The unique structure of LSM-trees, compared to traditional storage engines like B+ trees [7, 34], results in distinct write and read I/O costs, significantly impacting join performance.

7 CONCLUSION

In this study, we present a comprehensive configuration space for disk-based joins in LSM-tree data stores. We provide a thorough theoretical analysis of each component, which guides the design of our extensive benchmark. Through rigorous evaluation of diverse join methods under various working conditions, we derive several insights specified to LSM-tree and practical takeaways that challenge existing assumptions.

REFERENCES

[1] 2016. HBase. http://hbase.apache.org/. [Online; accessed 12-January-2022].
[2] 2021. WiredTiger. https://github.com/wiredtiger/wiredtiger.
[3] Ildar Absalyamov, Michael J Carey, and Vassilis J Tsotras. 2018. Lightweight cardinality estimation in LSM-based systems. In *Proceedings of the 2018 International Conference on Management of Data*. 841–855.
[4] Sattam Alsubaiee1 Yasser Altowim1 Hotham Altwaijry, Alexander Behm, Vinayak Borkar1 Yingyi Bu1 Michael Carey, Inci Cetindil1 Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova1 Raman Grover1 Zachary Heilbron, Pouria Pirzadeh1 Vassilis Tsotras7 Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: A Scalable, Open Source BDMS. *Proceedings of the VLDB Endowment* 7, 14 (2014).
[5] Apache. 2016. Cassandra. http://cassandra.apache.org.
[6] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1185–1196.
[7] Rudolf Bayer and Edward McCreight. 1970. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. 107–141.
[8] Jon Bentley. 1984. Programming pearls: algorithm design techniques. *Commun. ACM* 27, 9 (1984), 865–873.
[9] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.
[10] Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. 2018. Accordion: Better memory organization for LSM key-value stores. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1863–1875.
[11] Wei Cao, Feifei Li, Gui Huang, Jianghang Lou, Jianwei Zhao, Dengcheng He, Mengshi Sun, Yingqiang Zhang, Sheng Wang, Xueqiang Wu, et al. 2022. PolarDB-X: An Elastic Distributed Relational Database for Cloud-Native Applications. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2859–2872.
[12] Zhao Cao, Shimin Chen, Feifei Li, Min Wang, and X Sean Wang. 2013. LogKV: Exploiting key-value stores for event log processing. In *Proc. Conf. Innovative Data Syst. Res.*
[13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.
[14] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. 2021. Cosine: a cloud-cost optimized self-designing key-value storage engine. *Proceedings of the VLDB Endowment* 15, 1 (2021), 112–126.
[15] Guoqiang Jerry Chen, Janet L Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. 2016. Realtime data processing at facebook. In *Proceedings of the 2016 International Conference on Management of Data*. 1087–1098.
[16] CockroachDB. 2020. Pebble. https://github.com/cockroachdb/pebble.
[17] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2020. From wisckey to bourbon: A learned index for log-structured merge trees. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 155–171.
[18] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 79–94.
[19] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*. 505–520.
[20] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: granulating LSM-tree compactions correctly. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3071–3084.

[21] Dayan, Niv and Idreos, Stratos. 2019. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data*. 449–466.

[22] Giuseppe DeCandia, D Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, S Sivasubramanian A Pilchin, P Vosshall, and W Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. SOSP, 2007.

[23] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A Wood. 1984. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on management of data*. 1–8.

[24] Facebook. 2016. RocksDB. https://github.com/facebook/rocksdb.

[25] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1162–1175.

[26] Kevin P Gaffney, Martin Prammer, Larry Brasfield, D Richard Hipp, Dan Kennedy, and Jignesh M Patel. 2022. Sqlite: past, present, and future. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3535–3547.

[27] Google. 2016. LevelDB. https://github.com/google/leveldb/.

[28] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2019. DeepDB: Learn from Data, not from Queries! *Proceedings of the VLDB Endowment* 13, 7 (2019).

[29] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An optimized storage engine for large-scale E-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data*. 651–665.

[30] Wentao Huang, Yunhong Ji, Xuan Zhou, Bingsheng He, and Kian-Lee Tan. 2023. A Design Space Exploration and Evaluation for Main-Memory Hash Joins in Storage Class Memory. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1249–1263.

[31] Andy Huynh, Harshal A Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2021. Endure: A Robust Tuning Paradigm for LSM Trees Under Workload Uncertainty. *arXiv preprint arXiv:2110.13801* (2021).

[32] Andy Huynh, Harshal A. Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2023. Towards Flexibility and Robustness of LSM Trees. arXiv:2311.10005 [cs.DB]

[33] Andy Huynh, Harshal A Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2024. Towards flexibility and robustness of LSM trees. *The VLDB Journal* (2024), 1–24.

[34] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, et al. 2019. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn.. In *CIDR*.

[35] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S Kester, and Demi Guo. 2018. The data calculator: Data structure design and cost synthesis from first principles and learned cost models. In *Proceedings of the 2018 International Conference on Management of Data*. 535–550.

[36] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, et al. 2015. BetrFS: A Right-Optimized Write-Optimized File System.. In *FAST*, Vol. 15. 301–315.

[37] Christopher Jermaine, Alin Dobra, Subramanian Arumugam, Shantanu Joshi, and Abhijit Pol. 2005. A disk-based join with probabilistic guarantees. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 563–574.

[38] David Justen, Daniel Ritter, Campbell Fraser, Andrew Lamb, Allison Lee, Thomas Bodner, Mhd Yamen Haddad, Steffen Zeuch, Volker Markl, and Matthias Boehm. 2024. POLAR: Adaptive and Non-invasive Join Order Selection via Plans of Least Resistance. *Proceedings of the VLDB Endowment* 17, 6 (2024), 1350–1363.

[39] Ahmad Khazaie and Holger Pirk. 2023. SonicJoin: Fast, Robust and Worst-case Optimal.. In *EDBT*. 540–551.

[40] Taewoo Kim, Alexander Behm, Michael Blow, Vinayak Borkar, Yingyi Bu, Michael J Carey, Murtadha Hubail, Shiva Jahangiri, Jianfeng Jia, Chen Li, et al. 2020. Robust and efficient memory management in Apache AsterixDB. *Software: Practice and Experience* 50, 7 (2020), 1114–1151.

[41] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[42] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the join order benchmark. *The VLDB Journal* 27, 5 (2018), 643–668.

[43] Junfeng Liu, Fan Wang, Dingheng Mo, and Siqiang Luo. 2024. Structural Designs Meet Optimality: Exploring Optimized LSM-tree Structures in A Colossal Configuration Space. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.

[44] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 1–28.

[45] Chen Luo and Michael J Carey. 2019. Efficient data ingestion and query processing for LSM-based storage systems. *Proceedings of the VLDB Endowment* 12, 5 (2019), 531–543.

[46] Chen Luo and Michael J Carey. 2020. Breaking down memory walls: Adaptive memory management in LSM-based storage systems. *Proceedings of the VLDB Endowment* 14, 3 (2020), 241–254.

[47] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A robust space-time optimized range filter for key-value stores. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2071–2086.

[48] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoia, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2021. Flow-loss: Learning cardinality estimates that matter. *Proceedings of the VLDB Endowment* 14, 11 (2021).

[49] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking learned indexes. *Proceedings of the VLDB Endowment* 14, 1 (2020), 1–13.

[50] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* 14, 1 (2020), 1–13.

[51] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2020. Bao: Learning to steer query optimizers. *arXiv preprint arXiv:2004.03814* (2020).

[52] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*. 1275–1288.

[53] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: a learned query optimizer. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1705–1718.

[54] Ryan Marcus and Olga Papaemmanouil. [n.d.]. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proceedings of the VLDB Endowment* 12, 11 ([n. d.]).

[55] Ryan Marcus and Olga Papaemmanouil. 2018. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–4.

[56] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. Myrocks: Lsm-tree database storage engine serving facebook's social graph. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3217–3230.

[57] Scott Meyers. 2001. *Effective STL: 50 specific ways to improve your use of the standard template library*. Pearson Education.

[58] Dingheng Mo, Fanchao Chen, Siqiang Luo, and Caihua Shan. 2023. Learning to Optimize LSM-trees: Towards A Reinforcement Learning based Key-Value Store for Dynamic Workloads. *Proceedings of the ACM on Management of Data* 1, 3 (2023), 1–25.

[59] Ju Hyoung Mun, Zichen Zhu, Aneesh Raman, and Manos Athanassoulis. 2022. LSM-Trees Under (Memory) Pressure. In *Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*.

[60] MySQL. 2022. MySQL. *Web resource: https://www.mysql.com/* (2022).

[61] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.

[62] Arash Partow. 2013. General purpose hash function algorithms. (2013).

[63] Behandelt PostgreSQL. 2022. PostgreSQL. *Web resource: http://www.PostgreSQL.org/* (2022).

[64] Mohiuddin Abdul Qader, Shiwen Cheng, and Vagelis Hristidis. 2018. A comparative study of secondary indexing techniques in LSM-based NoSQL databases. In *Proceedings of the 2018 International Conference on Management of Data*. 551–566.

[65] Ibrahim Sabek and Tim Kraska. 2023. The Case for Learned In-Memory Joins. *Proceedings of the VLDB Endowment* 16, 7 (2023), 1749–1762.

[66] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An experimental comparison of thirteen relational equi-joins in main memory. In *Proceedings of the 2016 International Conference on Management of Data*. 1961–1976.

[67] Leonard D Shapiro. 1986. Join processing in database systems with large main memories. *ACM Transactions on Database Systems (TODS)* 11, 3 (1986), 239–264.

[68] Pradeep J Shetty, Richard P Spillane, Ravikant R Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. 2013. Building workload-independent storage with VT-trees. In *Presented as part of the 11th {USENIX} Conference on File and Storage Technologies ({FAST} 13)*. 17–30.

[69] Ji Sun and Guoliang Li. 2018. An End-to-End Learning-based Cost Estimator. *Proceedings of the VLDB Endowment* 13, 3 (2018).

[70] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1493–1509.

[71] Jing Wang, Youyou Lu, Qing Wang, Yuhao Zhang, and Jiwu Shu. 2023. Revisiting Secondary Indexing in {LSM-based} Storage Systems with Persistent Memory. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 817–832.

[72] Peizhi Wu and Gao Cong. 2021. A unified deep model of learning from both data and queries for cardinality estimation. In *Proceedings of the 2021 International Conference on Management of Data*. 2009–2022.

[73] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigümüs, and Jeffrey F Naughton. 2013. Predicting query execution time: Are optimizer cost models really unusable?. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 1081–1092.

[74] Ziniu Wu, Parimarjan Negi, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. 2023. FactorJoin: a new cardinality estimation framework for join queries. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[75] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: one cardinality estimator for all tables. *Proceedings of the VLDB Endowment* 14, 1 (2020), 61–73.