Columnstore and B+ tree – Are Hybrid Physical Designs Important?

Adam Dziedzic[†] Jingjing Wang[‡] Sudipto Das[§] Bolin Ding[§] Vivek R. Narasayya[§]

Manoj Syamala[§*]

[§]Microsoft Research
Redmond, WA, USA

[†]University of Chicago
Chicago, IL, USA

[‡]University of Washington
Seattle, WA, USA

ABSTRACT

Commercial DBMSs, such as Microsoft SQL Server, cater to diverse workloads including transaction processing, decision support, and operational analytics. They also support variety in physical design structures such as B+ tree and columnstore. The benefits of B+ tree for OLTP workloads and columnstore for decision support workloads are well-understood. However, the importance of *hybrid* physical designs, consisting of both columnstore and B+ tree indexes on the same database, is not well-studied — a focus of this paper. We first quantify the trade-offs using carefully-crafted micro-benchmarks. This micro-benchmarking indicates that hybrid physical designs can result in orders of magnitude better performance depending on the workload. For complex real-world applications, choosing an appropriate combination of columnstore and B+ tree indexes for a database workload is challenging. We extend the Database Engine Tuning Advisor for Microsoft SQL Server to recommend a suitable combination of B+ tree and columnstore indexes for a given workload. Through extensive experiments using industry-standard benchmarks and several real-world customer workloads, we quantify how a physical design tool capable of recommending hybrid physical designs can result in orders of magnitude better execution costs compared to approaches that rely either on columnstore-only or B+ tree-only designs.

KEYWORDS

Columnstore, B+ tree, Hybrid physical designs, operational analytics, hybrid transactional and analytical processing.

ACM Reference Format:
Adam Dziedzic, Jingjing Wang, Sudipto Das, Bolin Ding, Vivek R. Narasayya, Manoj Syamala. 2018. Columnstore and B+ tree – Are Hybrid Physical Designs Important?. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, Article 4, 14 pages. https://doi.org/10.1145/3183713.3190660

[*] Adam Dziedzic and Jingjing Wang performed the work while at Microsoft Research.

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3190660

1 INTRODUCTION

B+ tree indexes [6, 13] have been available in relational database systems (RDBMSs) for several decades and are widely used in practice. More recently, major commercial RDBMSs have also incorporated columnstore indexes [21, 22, 24, 33, 39]. Microsoft SQL Server supports both B+ tree and columnstore indexes on the same table, either as a *primary* index that contains data of all columns in the table, or a redundant *secondary* index with a subset of columns.

Commercial RDBMSs, such as SQL Server, support applications with workloads that vary from update-heavy OLTP, to read-heavy analytic and decision support workloads, to mixed workloads consisting of both OLTP and analytic queries on the same database for operational analytics scenarios. It is generally understood that columnstores are crucial to achieving high performance for analytic queries and that B+ tree indexes are key to supporting transactional workload efficiently. However, it is not well understood whether **hybrid** physical designs—*both* columnstore and B+ tree indexes on the same database and potentially the same table—are important for any of the above workloads.

To answer this question, we first empirically quantify the read and update characteristics of columnstore and B+ tree indexes using carefully-crafted micro-benchmarks on a commercial RDBMS—Microsoft SQL Server. We analyze performance across a range of important parameters such as data size, selectivity, query working memory, number of rows updated, and concurrency (Section 3).

For read-only queries, we find that both columnstore and B+ tree indexes can significantly outperform one another based on workload characteristics. B+ trees outshine columnstores when query predicates are selective even when all data is memory resident; and the trade-off shifts further in favor of B+ trees when data is not memory resident. Likewise, B+ trees can be a better option for providing data in sorted order when server memory is constrained. On the other hand, columnstores are often an order of magnitude faster for large scans whether or not the data is memory resident. For updates, B+ trees are significantly cheaper. Secondary columnstores incur much lower update cost compared to primary columnstore indexes, but are still much slower than B+ trees. This empirical study indicates that for certain workloads, hybrid physical designs can provide significant performance gains.

Despite the promise of hybrid physical designs, choosing an appropriate mix of B+ tree and columnstores for complex real-world workloads can be daunting even for expert DBAs. Motivated by this need, we extend Database Engine Tuning Advisor (DTA), a physical design tuning tool for SQL Server, to analyze and recommend both B+ tree and columnstore indexes when suitable for a given workload. We discuss the challenges in the design and implementation of our extensions to DTA and SQL Server to consider this expanded space of physical designs (Section 4). This new functionality in DTA was released in January 2017 as part of Community Technology Preview (CTP) release of Microsoft SQL Server 2017 [37].

Finally, we conduct extensive experiments using standard benchmarks, such as TPC-DS [40] and CH [12], and several real-world customer workloads (Section 5). We derive two major conclusions from our experiments: (*i*) hybrid physical designs can result in more than an order of magnitude lower execution costs for many workloads when compared to alternatives using B+ tree-only or columnstore-only; (*ii*) the extensions to DTA to recommend hybrid physical designs helps exploit the best of both worlds: selecting the appropriate combination of B+ tree-only, columnstore-only, or hybrid configurations appropriate for a given workload. Kester et al. [19] present a similar empirical study considering columnstore and secondary B+ tree indexes in a main-memory-optimized prototype system supporting shared scans, with the focus on concurrency. Our study considers a richer hybrid design space supported in a commercial-strength DBMS, with the focus on variety of workloads and an automated tool to recommend such hybrid designs.

To summarize, this paper makes the following contributions:

- We present an extensive experimental study using micro-benchmarks to systematically quantify the trade-offs associated with hybrid physical designs in a commercial RDBMS.
- We extend a commercial physical design tuning tool to add the ability to analyze and recommend hybrid physical designs based on the workload's characteristics.
- End-to-end experiments with several standard benchmarks and real-world customer workloads reveal that hybrid physical designs can result in orders of magnitude performance gains compared to B+ tree-only or columnstore-only designs.

2 PHYSICAL DESIGNS IN SQL SERVER

SQL Server supports a variety of physical design options, such as indexes, materialized views, and partitioning. In this paper, we focus only on the variety of indexes supported by SQL Server.

B+ tree and columnstore: RDBMSs have supported B+ trees and heap files for several decades. Since the advent of columnstores, which significantly outperform B+ trees for data analysis workloads, many commercial RDBMS vendors have added support for columnstore indexes (**CSI**). While the high-level design of columnstores is similar across different systems, there are many variations in what combination of indexes can be created, how they are built, compressed, maintained, and updated. In this section, we discuss the specific implementation of columnstore in Microsoft SQL Server.

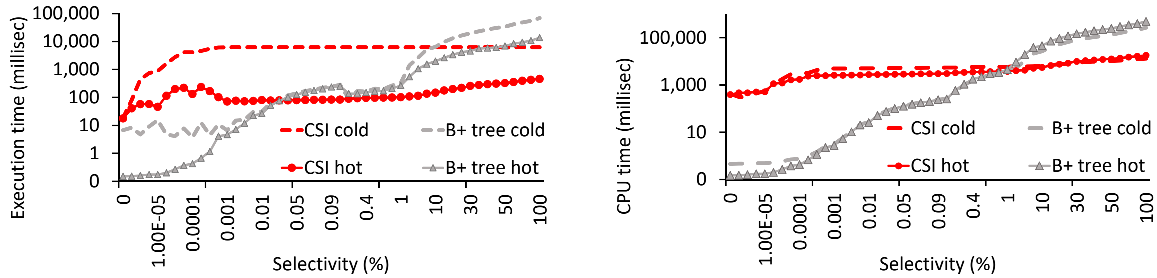
SQL Server supports columnstores as an additional mechanism to store data [23, 28]. Similar to B+ trees, columnstores in SQL Server are treated as indexes, which can either be *primary* (main storage of all columns of the table) or *secondary* (redundant storage with a subset of columns). SQL Server supports any combination of primary and secondary indexes on the same table. That is, the primary index can be a heap file, B+ tree, or a columnstore. A secondary index can be a B+ tree or a columnstore, with the restriction of a single columnstore index per table. B+ tree indexes provide ordering of data based on the key columns in the index and allow efficient lookups, while columnstores in SQL Server do not provide sort order and are optimized for efficient scanning. Columnstores allow vectorized operations on a dense array of homogenous types often on encoded values (called *batch mode* in SQL Server) [1, 7] which is significantly more efficient compared to row-at-a-time (or *row mode*) execution, typically used for B+ tree.

Compression: Columnstores support compression and query processing on compressed data [1, 2]. When building a columnstore, SQL Server selects a sort ordering of the columns that aims to maximize the compression ratio of the overall columnstore index. Columnstore index compression uses several encoding techniques, the most notable being dictionary encoding and run-length encoding [24]. Dictionary encoding converts data values from non-numeric domains (such as strings) to numeric domain. Run-length encoding compresses sorted runs (e.g., $2, 2, 2, \ldots, 2$ can be converted to 2, k repetitions). SQL Server's columnstores are comprised of sets of rows, called *row groups*. A row group contains between $100K - 1M$ rows, which are compressed independently. Each column in a row group forms a column segment. Primary and secondary columnstores use the same compression algorithms and have similar structure for compressed segments.

Updates: Inserts are handled via delta stores which are implemented as B+ trees [23]. Bulk loaded data is transformed directly into the compressed row groups. Smaller point updates are handled as a delete followed by an insert. Primary and secondary columnstores differ in how deletes are handled, driven by the application characteristics they optimize for. Secondary columnstores, optimized for operational analytics, have a delete buffer which is a B+ tree storing the logical row being deleted. When deleting a row, it is inserted into the delete buffer, allowing for fast logical deletion. However, query processing pays an additional overhead of an anti-semi join between the compressed row groups and the delete buffer. To reduce the cost of this anti-semi join, a background process periodically compresses the delete buffer into a *delete bitmap*, which stores the physical identifiers of the deleted rows, and eventually compacts the delete bitmap into the compressed segments. A primary columnstore on the other hand does not support a delete buffer, only the delete bitmap, which optimizes scan performance by avoiding the anti-semi join. Hence, deleting a row in a primary columnstore needs to scan the compressed row group to obtain the physical row locator, which is added to the delete bitmap. Primary columnstores are therefore more suitable for scans and bulks loads common in data warehouses, while secondary columnstores are amenable to small updates while still being efficient for scans optimized for operational analytics.

3 MICRO-BENCHMARKING HYBRID PHYSICAL DESIGNS

We use micro-benchmarks that allow us fine-grained control over data and queries to quantify the performance trade-offs between B+



(a) Execution time. (b) CPU time.

Figure 1: Execution and CPU time for hot and cold runs for a query with varying selectivity.

trees and columnstores and identify cases where the hybrid physical designs are crucial. We use the following broad categories of workloads in our performance study: (*a*) scans with single predicates with varying selectivity to study the trade-off between range scan of a B+ tree vs. columnstore scan; (*b*) sort and group by queries to study the benefit of the sort order supported by B+ tree; (*c*) update statements with varying numbers of updated rows to analyze the cost of updating the different index types; and (*d*) mixed workloads with different combinations of reads and updates.

3.1 Experimental setup

Hardware: All experiments were run on a server equipped with dual socket Intel ® Xeon ® CPU $E5-2660v2$ (10 cores per socket, 2 threads per core) clocked at 2.20 GHz, 64 KB L1 cache per core, 256 KB L2 cache per core and 25 MB L3 cache shared, 384 GB RAM, 18 TB HDD in RAID-0 configuration (with throughput of about 1 GB/sec for reads and 400 MB/sec for writes) and running Microsoft Windows Server 2012 R2 Datacenter (64 bit).

Software: We use a pre-release version of Microsoft SQL Server 2017 as the database engine.

Data set: We TPC-H [42] and other synthetically-generated data with sizes in range $1-100$ GB. Synthetic data set consists of tables with different numbers of columns. Each column contains uniformly distributed 32-bit integers in range from 0 to $2^{31}-1$ (similar to Kester et al. [19]).

Methodology: We execute the workloads and measure execution (elapsed) time, CPU time, memory usage and disk I/Os. We monitor query performance using the Query Store [29] and collect the system-wide performance statistics via Microsoft Windows Performance Monitor. Each experiment is run at least 5 times and we report the average of the collected data points.

3.2 Read-only queries

3.2.1 Impact of data skipping. Our first experiment studies the trade-off between range scans of B+ tree and full scan of columnstore. We use a 10 GB table with a single integer column. To control the amount of data accessed by the query, we use a simple query that selects a set of rows and computes an aggregate on it. We use the query (Q_1): `SELECT sum(col1) FROM table WHERE col1 < {1}` where the selectivity is controlled by setting the appropriate parameter for the predicate. We compare the performance of the query for a primary B+ tree vs. primary columnstore for both cold and hot runs. For cold runs, the data resides on HDD.

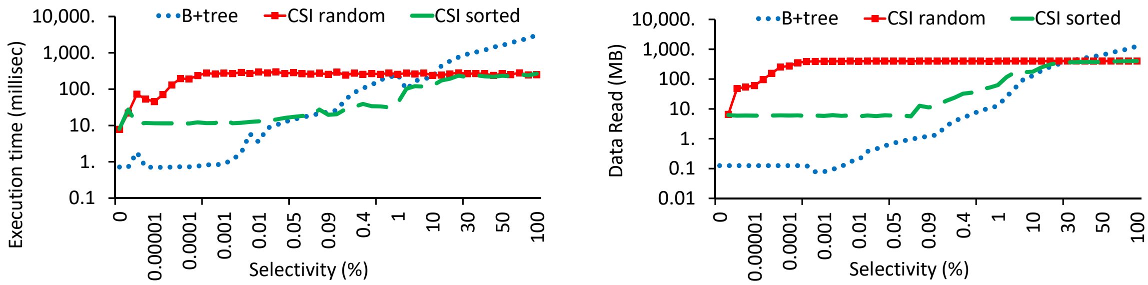
Figure 1 plots the execution time and CPU time (in ms, log scale) as we vary the selectivity of the predicate.[1] For low values of selectivity, B+ tree significantly outperforms CSI by about one to two orders of magnitude in execution time, and up to three orders of magnitude in CPU time. When selectivity is small, using a B+ tree implies few accessed pages where the optimizer chooses a single-threaded execution plan. Such sequential plans are more CPU-efficient compared to parallel plans, which are chosen in the case of CSI or for higher selectivity values for B+ tree (about 0.2% in our experiments). The change in degree of parallelism (DOP) from 1 to 40 at selectivity of 0.2% results in a dip in execution time (Figure 1(a)) and a jump in CPU time (Figure 1(b)). Note that for cold runs, when data needs to be accessed from storage, the benefits of B+ tree is more significant since it accesses significantly less data when the query has low selectivity. The extent of this benefit depends on the bandwidth and access latencies of the storage media—the slower the storage, the more pronounced the benefit of B+ tree is. For cold runs, the crossover point for execution time is 10%.

Note that columnstores also benefit from smaller amounts of data accessed by very selective queries. SQL Server stores simple aggregates (min and max) for each column in each segment which allows data skipping (or *segment elimination*) if the segment is guaranteed to not contain data relevant to the query [23, 30]. Several approaches have been proposed in literature to aid such data skipping. For instance: (*a*) sorted columnstores, such as projections in C-Store and Vertica [39, 43]; or (*b*) small materialized aggregates (e.g., min, max, sum, count) for each column segment [30].

We now study how columnstores compare with B+ tree if they can skip data more aggressively. SQL Server does not provide a sort order guarantee on a specific column in a CSI. However, if data was pre-sorted on a specific column C_1 when a CSI was built, the range of value in different segments of C_1 will be sorted, which can be used to eliminate irrelevant segments if there is a predicate on C_1.

To achieve this behavior for Q_1 where the predicate was on `col1`, we sort the data on `col1` before building a CSI and compare the performance of CSI when it is built on data in random order vs. sorted order on `col1`. Figure 2 reports the execution time and

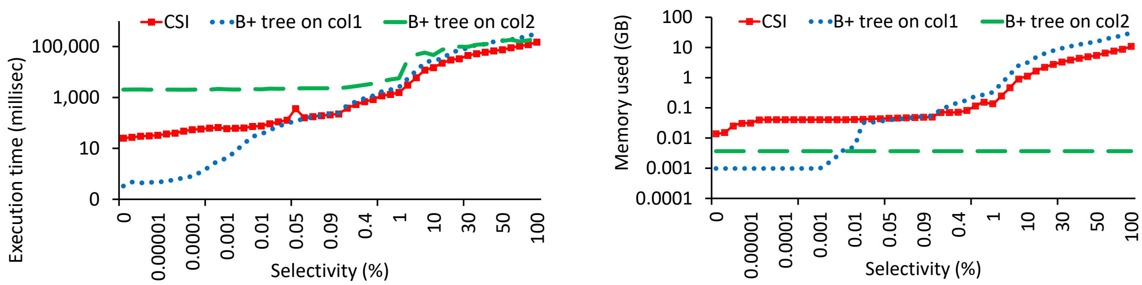
[1] We use selectivity to denote the fraction of rows in the table that qualify the predicate, i.e., higher selectivity implies more rows qualify.



(a) Execution time.

(b) Data read.

Figure 2: Execution time and amount of data read for B+ tree and CSI (sorted and unsorted) for a query with varying selectivity.



(a) Execution time.

(b) Query memory used (GB).

Figure 3: Execution time and amount of memory used for B+ tree and CSI for a query with varying selectivity on one column (col1) and sort order on another column (col2).

the amount of data read (in MB) for a cold run as we vary the selectivity. As expected, the sorted ranges allow more segments to be skipped, making the CSI more competitive with B+ tree. Referring to Figure 2(a), the crossover selectivity moves to 0.09% for sorted CSI (compared to about 10% for the CSI with random data). As is evident from Figure 2(b), the sorted CSI accesses one to two orders of magnitude less data compared to unsorted CSI. Note, however, that the data access crossover is around 10%, which implies that with a CSI, the query latency is comparable to B+ tree even when an order of magnitude more data is being accessed. This efficiency can be attributed to vectorized processing in CSI as well as other optimizations such as accessing and prefetching larger data blocks (megabytes in CSI compared to kilobytes in B+ tree). In Appendix A.1, we present the graph for CPU Time.

3.2.2 Impact of sort order. B+ tree indexes also provide sort order on the key columns in the index. Such sort order is beneficial if a query result requests a sort order, or in execution plans that can benefit from sorted data order, such as using a streaming aggregate instead of a hash-based aggregate, or a merge join instead of a hash join. In all such cases, not having to sort or hash the data reduces the memory required for the query execution. CSI's in SQL Server provide CPU and I/O-efficient data processing, but do not provide sort order. While it is possible to have sorted columnstores (such as projections in Vertica [43]), maintaining the sort order in the presence of arbitrary updates becomes expensive.

Explicit sort order. We first consider a query which requests explicit sort order on a column, while having a predicate on another column. Q_2: `SELECT col1, col2 FROM table WHERE col1 < {1} ORDER BY col2`. The table has two integer columns with 10 GB data and all data is memory-resident during query execution. We consider three physical designs: (*a*) Primary CSI where scan, filter, and sorting the result will be performed during query execution. (*b*) Primary B+ tree keyed on `col1`, with `col2` as included column. Here B+ tree range scan is based on the filter, though the result must be sorted during query execution. (*c*) Primary B+ tree keyed on `col2`, with `col1` as included column. Here the filter is evaluated during query execution after scanning data in sort order.

Figure 3 presents the execution time (Figure 3(a)) and the memory used by the query during execution (Figure 3(b)). Since scanning CSI is significantly more efficient than scanning B+ tree, option (c) is the most expensive in terms of execution time, but also has low memory footprint since no sorting of data is required. On the other hand, when selectivity is low, option (b) allows efficient access to data by touching very little data, and since the result size is small, the cost to sort the result is also small. Compared to option (a), the benefits of efficient data selection of B+ tree dominates. However, as the selectivity increases, and more data needs to be processed, the benefits of the efficient CSI scan and sort starts to dominate, and hence eventually CSI outperforms both the B+ tree-based options for selectivity values above 1%. Therefore, when accessing large amounts of data, the sort order of B+ tree does not provide benefits

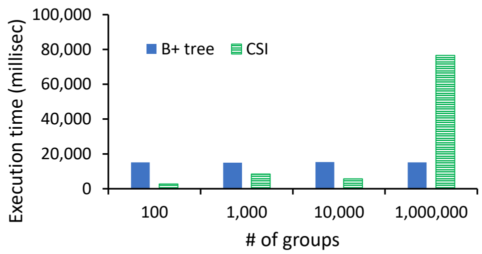


Figure 4: Execution time for group by query where we vary the number of groups.

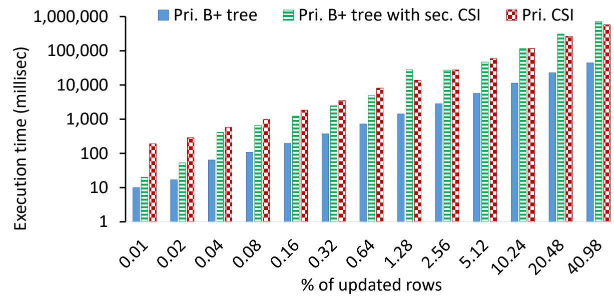


Figure 5: Execution time for update statements that update different number of rows.

above CSI, especially when sufficient memory is available to sort the data in-memory during query execution.

Sort order benefiting execution. We next consider a query to study how the sort order provided by key columns in a B+ tree can benefit query execution when the query does not explicitly require a sort. We consider the case of aggregation where a streaming aggregation can be used when data is sorted, as opposed to a hash-based aggregate. We consider the query Q_3: `SELECT col1, sum(col2) FROM table GROUP BY col1`. We use a table with 20 GB data, two integer columns, where we vary the number of distinct values of `col1` from $100 - 1,000,000$. We report results from a hot run. To study performance when there is insufficient query working memory, we limit the query's working memory (called grant memory in SQL Server) in this experiment. For cases where the number of groups is large, and a hash-based aggregation is used, disk-based aggregation is used when memory is insufficient.

Figure 4 presents the execution time of the query as we vary the number of groups and compares the performance of a primary B+ tree (on `col1`) with that of a primary CSI. For smaller number of groups, where the hash-based aggregation can be performed in memory, CSI significantly outperforms B+ tree due to two reasons: (a) efficient scan and vectorized execution; and (b) compression achieved by CSI for cases where the number of distinct values of `col1` is small, resulting in CSI accessing much less data compared to B+ tree which cannot benefit from such compression. However, as the number of distinct values of `col1` increases, the benefits of compression decreases. Moreover, the memory requirement for the hash-based aggregation also increases. When this memory requirement is higher than the working memory for the query, a disk-based aggregation implementation makes the CSI significantly slower compared to B+ tree where the sort order allows streaming aggregate which has very low memory requirement. An approach such as the incremental spilling with replacement selection [15] can potentially be used to improve performance for such cases.

3.2.3 Key Findings. B+ trees are important for queries with very selective predicates (in our experiments, less than 0.7% for the memory-resident data and less than 10% for the disk-resident data). The crossover point depends on the access latency and bandwidth of the data storage medium—the slower the storage, the higher is the cross-over point. Data sort order in B+ tree is beneficial only when memory for sorting or computing hash-based aggregate is scarce. If operations can be completed in memory, then columnstores result in significantly better (about 5× in our experiments) performance compared to B+ trees. However, if memory is insufficient and a disk-based implementation of hashing or sorting is used, then sortedness of data in B+ tree helps it achieve significantly better performance (up to 5× in our experiments) compared to columnstore.

3.3 Updates

We now analyze the cost of updating the B+ tree and columnstore indexes for updates of different sizes, an experiment modeled along the lines of Larson et al. [23]. We use the update statement Q_4: `UPDATE top (N rows) SET l_quantity +=1, l_extendedprice += 0.01 WHERE l_shipdate = '{1}'` on TPC-H 30 GB. We report results from a hot run with a single thread issuing updates. As discussed in Section 2, primary and secondary CSI in SQL Server process updates differently. Therefore, we consider three different types of physical designs: B+ tree, secondary, and primary CSI. We use a primary B+ tree on `l_shipdate` for the B+ tree-only and the design with secondary CSI.

Figure 5 reports the execution time for the statements as we vary N, the number of rows which are updated. As expected, the cost to update B+ tree is the cheapest. Updates in CSIs are handled with internal structures composed of B+ trees. For small updates (i.e., 0.01% of the data), a secondary CSI is about 2× slower compared to a just updating the primary B+ tree. However, since to update (which is a delete followed by an insert) a primary CSI, deletes need to be added to the delete bitmap (see Section 2), there is a high cost to locate the deleted rows in the column segments so that its physical locator can be added to the delete bitmap. This makes updating the primary CSI significantly more expensive compared to both B+ tree-only or secondary CSI.

As the percentage of the updated rows increases, the performance for the secondary CSI deteriorates in comparison to the primary B+ tree and is similar to the performance of the primary CSI. When about 40% of data is updated, both columnstore indexes are about 16× slower than the B+ tree. A secondary CSI is faster than primary CSI when a small amount of data is updated and similar to a primary CSI when more than about 1% of data is updated.

3.4 Mixed workload

Many applications execute a mix of OLTP and data analysis queries in an operational system to get quick operational insights from data. In this section, we mimic such a setup where we have two query types: an update statement which is Q_4 from Section 3.3 and a select query Q_5: `SELECT sum(l_quantity) sum_quantity,`

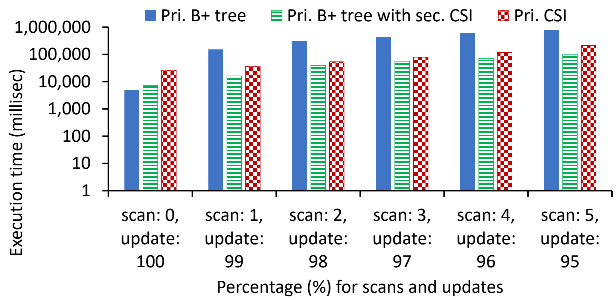


Figure 6: Execution time for mixed workload executed on three different physical designs.

`sum(l_extendedprice * (1-l_discount)) FROM lineitem WHERE l_shipdate between '1' and DATEADD (day, 1, '1')`. For Q_4, we set N to 10. We use multiple threads (10 for this experiment) to issue these requests and record the execution time. When executing concurrent read and write transactions, the isolation level has significant influence on lock contention. We use SQL Server's default isolation level Read Committed.

We report performance for three different physical designs: (*A*) a primary B+ tree on `l_orderkey` and `l_linenumber` and a secondary B+ tree on `l_shipdate`; (*B*) a primary B+ tree on `l_orderkey` and `l_linenumber` and a secondary B+ tree on `l_shipdate` and a secondary CSI on all columns; (*C*) a primary CSI and a secondary B+ tree on `l_shipdate`. In all three cases, the secondary B+ tree on `l_shipdate` helps with the selective predicate for Q_4.

Figure 6 reports the average workload execution time as we change the percentage of updates from 100% (with no scans) to 95% (with 5% of scans), reducing in steps of 1%. Updates are small and short running transactions while scans are long-running and resource-intensive analytical queries. For a given thread, we randomly select a scan or update to be executed with probability depending on the specified percentage. An update is executed for a randomly-chosen shipping date and the top 10 lineitems are modified. Note that even at 5% of the workload, the scans dominate the updates in terms of resources consumed, thus in practice making this workload scan-heavy.

When there are no scans, the performance of B+ tree is superior in comparison to the CSI (similar to Section 3.3). However, even for the small percentage of scans of 1%, the CPU-efficiency of CSI in speeding up the scans helps improve the average workload execution cost, even though there is a small increase in the execution time of Q_4. Option (B) has the best performance, since secondary CSI strikes a right balance between increased overhead of small updates vs. improved efficiency for large scans when compared to a B+ tree-only design. This experiment provides evidence that a hybrid physical design with B+ tree and CSI can provide significant performance boost for mixed workloads.

3.5 Key takeaways

We summarize the key findings of our micro-benchmarking study in Table 1 where we identify which physical design option (among B+ tree primary CSI, and secondary CSI) is suitable for a specific type of query pattern. We differentiate between primary and secondary CSI (unlike B+ tree) due to their difference in update characteristics. In

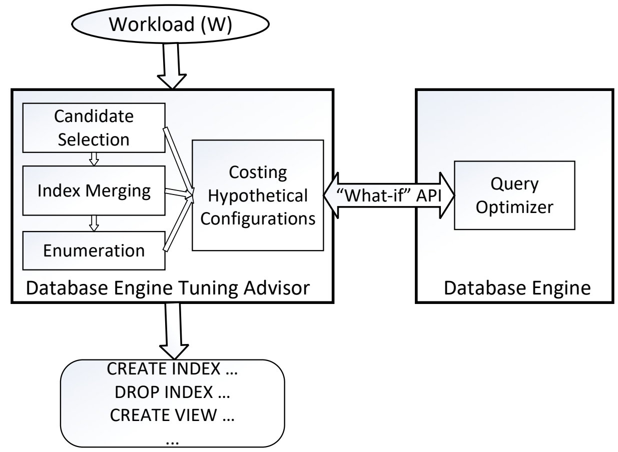


Figure 7: An architectural overview of DTA.

a nutshell, B+ tree indexes are suitable for short range scans where the index allows efficient point and short range lookups. B+ trees are also the cheapest to update. On the other hand, primary CSIs are most suitable for large scans and bulk updates typical in data warehousing and analysis workload. Secondary CSIs can provide significant speed-up for operational analytics on the same database where the OLTP application generating the data also runs. The basic workload axes can be combined in a variety of ways where a mix of the basic physical design axes are needed for optimal performance.

4 RECOMMENDING HYBRID DESIGNS

The empirical results of Section 3 highlight the relative strengths of B+ trees and columnstores and indicates the potential of combining them for certain workloads. However, choosing an appropriate physical design for a workload can be a difficult problem, even for expert DBAs. Even before columnstore indexes were introduced, several commercial DBMSs developed industry-strength *physical design tuning advisors* that can automatically recommend a good mix of physical design structures (e.g. B+ tree indexes, materialized views) for a given workload of SQL queries and updates [4, 14, 44].

Microsoft SQL Server ships Database Engine Tuning Advisor (DTA) [4, 9] to help analyze the complex space of physical design choices. DTA can recommend B+ tree indexes (primary and/or secondary), materialized views, and partitioning in one holistic search and costing framework. We extended DTA to analyze the *combined* space of B+ tree and columnstore indexes. By analyzing the workload, DTA is now capable of recommending B+ tree indexes only, columnstore indexes only, or a combination. This version of DTA was released in January 2017 as part of Community Technology Preview (CTP) releases of Microsoft SQL Server 2017 [37]. In this section, we: (*a*) highlight some of the challenges that arise when incorporating columnstore indexes into physical database design, and (*b*) outline our solution by describing key changes in DTA. We begin by first providing a brief overview of the architecture of DTA.

4.1 DTA Architecture

Given a user-specified workload $\mathcal{W}$ (which is a set of SQL statements with associated weights), DTA performs a cost-based search

Physical design \ Workload	**Short scans**	**Large scans**	**Short updates**	**Large updates**
B+ tree-only	most suitable	least suitable	most suitable	most suitable
Primary CSI-only	medium	most suitable	least suitable	least suitable
Secondary CSI with B+ tree	least suitable	medium	medium	least suitable

Table 1: Summarizing the key results of micro-benchmark study in terms of the basic axes for workload and physical design. We assume all secondary indexes are covering.

to identify a set of physical design changes that will minimize the total optimizer-estimated cost of $\mathcal{W}$ subject to constraints such as the total storage budget. Figure 7 provides an overview of DTA's architecture and some key components. Here we focus on components that are necessary to understand our extensions in DTA to support hybrid physical designs that include columnstore indexes; readers can refer to Agrawal et al. [4] and Chaudhuri et al. [9] for more details. Even though DTA can recommend materialized views and partitioning, for ease of exposition, we only focus on indexes.

The first stage in DTA is a local per-query analysis referred to as *candidate selection* where DTA analyzes each query $Q \in \mathcal{W}$ to determine the optimal set of indexes. Once the optimal set of indexes is identified for each Q, DTA performs a global workload-level analysis stage. The first step in global analysis is *index merging* which explores the potential to merge indexes on the same table [11]. Subsequently, DTA performs a global search over all indexes (union of candidate and merged indexes) and queries in $\mathcal{W}$ to find the set of indexes which will minimize the total cost of $\mathcal{W}$ subject to the specified constraints.

DTA uses a cost-based search – its objective is to find the configuration with the lowest optimizer-estimated cost for the workload that meets the specified constraints. To achieve costing for indexes which are not yet built, DTA uses a *"what-if"* API to simulate *hypothetical indexes*, which are metadata entries on the server sufficient for the optimizer to generate an estimated plan which will be used if the indexes were built [10]. For a given a query Q and a configuration C, this API returns the estimated query plan (and its cost) the optimizer will use if that configuration were to be materialized.

4.2 Extensions to "What-If" API

To compile an execution plan with hypothetical indexes, the optimizer needs index metadata (e.g., columns in the index), number of rows, and index size to determine the cost of accessing the relevant pages in the index. For B+ tree indexes, all columns part of the index are stored co-located on the leaf pages. Thus, if the optimizer considers a B+ tree index, the number of pages in the index which are relevant to answer the query is independent of the number of columns needed by the query. However, since columnstore indexes are stored column-at-a-time, the execution engine needs to only access the columns relevant to the query. Therefore, the optimizer needs the *per-column sizes* for columnstore indexes to estimate the cost of accessing a columnstore index for a given query.

We added two extensions to the query optimizer of SQL Server to consider columnstore indexes in the "what-if" mode. First, we augmented the engine to support creating the relevant metadata for hypothetical columnstores. This extension allows the optimizer to recognize these hypothetical indexes as columnstore indexes to enable the same set of search and transformation rules as materialized columnstores. Second, we augmented the optimizer's "what-if" API to add the ability to specify per-column sizes for columnstore indexes. This extension is useful for considering both existing and hypothetical columnstores in the "what-if" mode.

4.3 Search Space with Columnstore indexes

We added the ability in DTA to optionally recommend columnstore indexes in conjunction with all other physical design recommendations that DTA already supports. We support recommending both primary and secondary columnstore indexes on a table.

Candidate Selection. The first stage is to identify candidate columnstore indexes during DTA's candidate selection which analyzes individual queries. We consider columnstore indexes only on tables referenced in the query. SQL Server has limitations on several column data types which cannot be included in a columnstore index. We use the database schema information to determine which columns can be included in a columnstore index. Since we support both primary and secondary columnstore index recommendations, this data type limitations influences what kind of columnstore index the DTA can recommend. For instance, if a table has a column with a data type which is not supported by columnstore indexes, we cannot build a primary columnstore index on that table since a primary index must include all columns. We consider a candidate secondary index by excluding the unsupported column types.

As of writing, SQL Server supports only one columnstore index per table. This constraint influences the choice of CSI candidates. There are two alternatives for which columns to include in the CSI candidate: (*i*) only include the columns that were referenced in the workload's SELECT statements; and (*ii*) include all the table's columns whose types can be included in a CSI. While our algorithm can support option (*i*), we chose option (*ii*) in our implementation. This is partly because if a column is not accessed by the query, the execution engine does not need to access those columns. Hence, unlike having wider B+ tree indexes, having additional columns in a CSI does not impact query execution cost. Furthermore, with option (*i*), we will have to build the widest columnstore index that includes all columns in a table which have been referenced in the workload, which in many cases turned out to be all columns in the table. Moreover, by including all columns in the columnstore index candidate, it is still useful for ad-hoc queries which may reference other columns in the table. Note, however, that this design choice could increase the maintenance cost for these CSIs if the table is frequently updated. The workload-level search considers this maintenance cost. The CSI candidates are generated in addition to any B+ tree index candidates generated by DTA's existing algorithm used for B+ tree indexes.

Once the set of candidate columnstore and B+ tree indexes are generated, DTA creates the necessary hypothetical indexes for the candidates (if not already created for another query), and then leverages the "what-if" API to determine which subset of indexes are referenced by the optimizer and the query's cost in the referenced configuration. No further changes are needed in the rest of DTA's candidate selection algorithm.

Workload-level Search. Once the candidate indexes are identified, the next stage is to search through the alternative configurations to determine which indexes are beneficial to the workload. Since columnstore and B+ tree cannot be merged, and we are considering one columnstore with all allowed columns, when merging two indexes, if at least one of the indexes in a columnstore, then the candidates are not merged. After merging, the global search finds the configuration that reduces the total workload cost. The only changes in this stage are: (*a*) any configuration with a columnstore index must restrict to one index per table; and (*b*) when costing configurations with a columnstore, we need to estimate per-column sizes, and use the extended "what-if" API for costing.

4.4 Columnstore Size Estimation

In order to cost a query using the "what-if" API for a configuration consisting of a columnstore index, we need to provide the size of *each column* in that index. To support a user-specified constraint of maximum storage bound for the recommendation, we need to estimate the total size of an index. Therefore, one of the requirements is to estimate the per-column sizes of a hypothetical columnstore index, i.e. *without* building the index. Stated more precisely, given a table T with C columns and N rows, currently stored in a row-store format (either a B+ tree or a Heap file), we need to estimate the per-column size of the columnstore index on the table.

There are two main challenges in columnstore size estimation. First, for scalability of DTA for large tables, we cannot afford to scan and execute the encoding algorithms on the entire data. Therefore, we need techniques to estimate the size of the index using samples of the data obtained using *block-level* samples. Using block-level sampling has one significant limitation. If the data in the blocks are sorted by one or more columns (which is the case for B+ tree indexes), then selecting all rows in a sampled page introduces bias in the samples due to correlations. To correct for this bias, we use the block-level sampling technique described in Chaudhuri et al. [8]. Second, when SQL Server builds a columnstore index, it applies a combination of encoding techniques to compress data as described in Section 2. The choice of the encoding techniques, and therefore the resulting compression ratio is dependent on the data types and distribution [24]. Hence, we need techniques to estimate the size of the *compressed* representation of the column. Below we describe two size estimation techniques using samples.

Figure 8 illustrates the run-length encoding algorithm used to compress columnstores in SQL Server using a simple example with two integer columns. The example also illustrates some of the challenges we face in our size estimation. First, the size of encoded data is dependent on the number of runs, which is again dependent on data distributions of individual columns. Second, to achieve long runs, SQL Server also sorts the data, starting with the least distinct column (column B in Figure 8(b)). Third, the number of runs of other columns now depend on the joint distribution of the columns ($\langle B, A\rangle$ in Figure and 8(c)). Last, we need to estimate all these aspects with just a sample and with several approximations to keep the overheads of size estimation within reasonable bounds.

A	B
3	0
3	1
0	0
1	0
3	1
3	1

(a) Original data

A	B
3	0
0	0
1	0
3	1
3	1
3	1

(b) Sorted by B

A	B
0	0
1	0
3	0
3	1
3	1
3	1

(c) Sorted by B, A

A	B
0, 1	0, 3
1, 1	1, 3
3, 4	

(d) Encoded segments

Figure 8: Example of Run-length encoding used to compress data in columnstores in SQL Server.

Black-box approach One approach is to first build a columnstore index on the *sample* and then for each column, scale up the size of the column in the index by the inverse of the sampling ratio. This approach treats the compression logic as a black-box and assumes that compressed columns size scales up linearly with sample size. The advantage of the black-box approach is its simplicity and that it requires no changes even when the compression algorithm in the engine changes. On the other hand, its accuracy can suffer since the above linearity assumption often does not hold. Consider for example columns with very few unique values, such as `n_nationkey` in TPC-H benchmark [42], which has only 25 distinct values. Any foreign-key column that references `n_nationkey` can have at most 25 distinct values in that column. Therefore, every row group of the columnstore index can have at most 25 distinct values, whereas this estimator would significantly overestimate the size. Further, creating a columnstore index on the sample incurs relatively high overhead with potentially multiple sorts (necessary to run the compression algorithm) and the cost of persisting the index. The next approach attempts to overcome these limitations.

Modeling Runs using Distinct Value Estimation As described earlier, columnstore indexes in SQL Server use run-length encoding to compress data [23]. A run is a maximal sequence of identical values. The effectiveness of run-length encoding depends on the number of runs in the column and the length of each run. Consider the special case of a table with a single column. When data is sorted, it results in the fewest number of runs, which equals the number of distinct values in that column. Considering the example in Figure 8, if we sort the table on $\langle B, A\rangle$, where B is the major sort column and A is the minor sort column, then the number of runs in column A is *at most* equal to the number of distinct combinations of $\langle B, A\rangle$. The figure shows an example where the number of runs in column A is 3 even though the number of distinct combinations of $\langle B, A\rangle$ are 4.

SQL Server uses a greedy strategy that picks the next column to sort by based on the column with the fewest runs; we mimic this approach in our technique. One approximation we make for efficiency is that we use the distinct number of combinations of columns (which is an upper bound on the number of runs as described above) as the basis of our greedy step. For estimating the number of distinct values for a given set of columns, we adapt the GEE estimator [8]. The GEE estimator only scales up the number of *small* groups (i.e., groups that occur only once in the sample) in the sample by the inverse of the sampling ratio. Other groups (i.e., values that occur more than once in the sample) are only counted once. The advantages of this approach compared to the black-box approach are: (*a*) It is more efficient since it does not incur the cost of sort(s) of the sampled data or writing index data. (*b*) Despite the inherent hardness of estimating number of distinct values using a sample, in practice this approach is often more accurate.

4.5 Future extensions

Variants of columnstore indexes: Many other commercial DBMSs also support columnstores which differ in design and implementation from the SQL Server's columnstore. While, our discussion in this section focused on the changes made to DTA that are specific to SQL Server's support for columnstores, DTA's framework is extensible to many variants in columnstore technologies. For instance, Vertica supports projections [22, 43] which allow column ordering, thus providing an explicit sort order for columns in the columnstore. Since DTA already supports the ability to leverage any sort requirements of a query and uses it to determine the sort order for B+ tree indexes, extending support in DTA to consider sort order in columnstore indexes is straightforward – candidate selection needs to be aware of sort requirements in a query to determine an appropriate sort order. If multiple columnstores are allowed on the same table, then similar to B+ tree, candidate selection and merging needs to be extended to support multiple columnstore candidates.
Columnstore size estimation: Efficiently and accurately estimating the size of a columnstore will play a crucial role in improving the quality of recommendations for hybrid physical designs. The sub-problem of efficiently estimating the number of runs in a column efficiently (e.g., with a limited number of sorts of the sample) remains open. Further, modeling aspects such as each row group is compressed independently, could also improve accuracy.
Impact on query optimizer and execution: The use of B+ tree and columnstores for the same query also presents interesting challenges for query optimization and execution. For instance, the optimizer's search space is much larger, thus requiring heuristics to prune the search space to keep optimization times within reasonable bounds. Moreover, data stored in columnstores are amenable to vectorized (or *batch mode* in SQL Server) processing, while B+ tree indexes typically use *row-at-a-time* (or *row mode* in SQL Server) processing. Thus, considering B+ tree indexes and columnstores when optimizing a given query implies the optimizer needs to estimate the costs in these different execution modes. Last, columnstores have very different locking characteristics compared to B+ tree indexes, which impact query execution as well, aspects which are often out-of-model for the query optimizer. These introduce novel challenges in modeling the execution of hybrid physical designs.

5 END-TO-END EVALUATION

In Section 3, we used several micro-benchmarks to demonstrate the design space of hybrid physical designs. In this section, we use industry-standard benchmarks and several real-world customer workloads to evaluate whether hybrid physical designs help improve query performance. Furthermore, for such complex workloads, we also evaluate the effectiveness of our extensions to DTA in finding these hybrid physical designs.

The key takeaways from the experiments in this section are: (*i*) Hybrid physical designs help leverage the best of both B+ tree and columnstore indexes. In many complex workloads, hybrid physical designs can result in one to two orders of magnitude improvement in execution costs compared to columnstore or B+ tree-only designs. Note that there are also workloads where columnstore-only (e.g., TPC-H [42]) or B+ tree-only (e.g., TPC-C [41]) are sufficient. (*ii*) Extensions to DTA that analyze and recommend hybrid physical designs can help find the appropriate set of B+ tree and columnstore indexes based on characteristics of the workload. The benefit of DTA's extensions is that this decision can be automated, cost-based, and workload-dependent. (*iii*) There are additional challenges in query optimization to find the optimal plan (in terms of execution cost) as well as in concurrency and locking which needs to be considered to leverage the best of hybrid physical designs, aspects which are potential avenues for future work.

5.1 Experimental Setup and Workloads

We use the same hardware and software setup as described in Section 3. We use DTA to analyze the queries to identify an appropriate set of indexes. We consider three alternative physical designs: (*a*) **B+ tree**-only, where DTA is used to find an appropriate set of B+ tree indexes; (*b*) **columnstore**-only, where a secondary (non-clustered) columnstore is built on all tables in the database; and (*c*) **hybrid**, where DTA is used to identify the appropriate set of B+ tree and columnstore indexes for the queries.

Our experiments use workloads from two categories: (*a*) read-only workload comprising a set of read-only queries common in data analysis and decision support workloads; and (*b*) mixed workloads with both OLTP and decision support workloads executing on the same database, similar to operational analytics or HTAP.

For read-only, we use industry-standard TPC-DS benchmark [40] and five real customer workloads. The customer workloads represent several decision support workloads from five different customers of SQL Server. Table 2 reports some aggregate statistics about the schema of these read-only workloads, such as the database size, no. of tables, maximum table size, and average number of columns per table. The table also provides some statistics about the complexity of the queries in terms of the number of joins per query and the number of physical operators that appear in an execution plan chosen by the query optimizer for a given query. As evident from the table, these customer workloads represent complex queries over diverse schemas and database sizes.

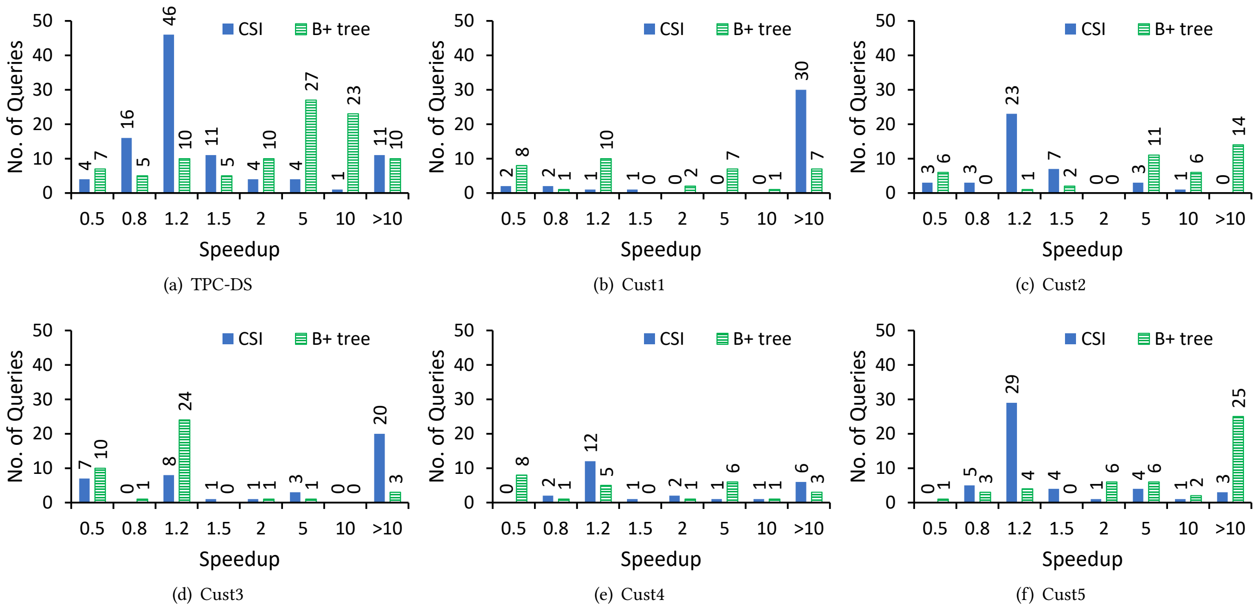
An emerging workload pattern is where the transactional database is also used for analysis and insights, often resulting in mixed OLTP and decision support queries executing on the same database. We use the CH benchmark [12] as a representative of this pattern. The CH benchmark is an extension of the TPC-C benchmark and schema with three additional tables and 22 additional queries (modeled along the TPC-H queries). The queries are designed to answer different business questions on the TPC-C transactional data.

5.2 Execution Cost Improvements

5.2.1 Read-only Workload. We use DTA to identify the appropriate indexes for each query in a workload, implement the indexes, and execute the query ten times. We report our results based on the median. The queries execute warm, and the server has sufficient memory to hold the entire working set in memory.

Workload	DB size	# tables	Max table size	Avg. # cols	# queries	Avg. # joins	Avg. # ops per plan
TPC-DS	87.7 GB	24	34.9 GB	17.2	97	7.9	28.2
Cust1	172 GB	23	63.8 GB	14.1	36	7.2	29.1
Cust2	44.6 GB	614	44.6 GB	23.5	40	8.1	28.3
Cust3	138.4 GB	3394	79.8 GB	26.3	40	8.75	24.1
Cust4	93 GB	22	54.8 GB	20.32	24	6.9	24.4
Cust5	9.83 GB	474	1.52 GB	5.5	47	21.6	53.3

Table 2: Aggregate statistics about the schema and query complexity for the read-only workloads.



(a) TPC-DS (b) Cust1 (c) Cust2

(d) Cust3 (e) Cust4 (f) Cust5

Figure 9: Distribution of speedup factor (for CPU time) achieved by a hybrid physical design compared to columnstore-only (CSI) and B+ tree-only designs.

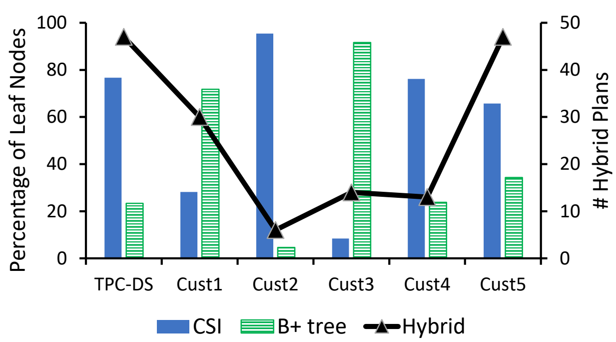


Figure 10: Summary of indexes chosen in the query plans. CSI and B+ tree correspond to the percentage of leaf nodes which are accessing columnstore and B+ tree respectively. The figure reports the average over all queries in the workload. Hybrid is the number of queries where both columnstore and B+ tree indexes were used in the execution plan the optimizer chose.

We use the amount of CPU time consumed by the queries as a measure of execution cost, since CPU time is dependent on the logical amount of work done by the query. We use SQL Server's Dynamic Management Views to obtain a query's CPU time.

Figure 9 plots the distribution of Speedup (in CPU time) obtained with a hybrid physical design as compared to the columnstore-only (CSI) and B+ tree-only physical designs. We compute the speedup of hybrid compared to CSI as: $\frac{CPUTime_{CSI}}{CPUTime_{hybrid}}$, and similarly for B+ tree. Therefore, a speedup > 1 implies hybrid is cheaper in execution, while < 1 implies hybrid is more expensive.

As is evident from Figure 9, hybrid leverages the best of columnstore and B+ tree across several workloads. For each workload, there are several queries for which a hybrid physical design results in more than an order of magnitude improvement in execution cost. In some cases, the improvement is $2 - 3$ orders of magnitude.

For the TPC-DS workload, there are 11 queries where a hybrid design results in more than an order of magnitude reduction in execution cost compared to columnstore-only. Moreover, there are 20 queries with $1.2\times - 10\times$ improvement. The improvements of hybrid compared to B+ tree is even more pronounced, primarily due to superior performance of CSI over B+ tree-only configurations.

The benefits of hybrid designs are also evident across several real-world customer workloads as well. For instance, for Cust1 and Cust3, hybrid results in more than an order of magnitude reduction in execution costs for a significant fraction of the workload when compared to CSI. On the other hand, for Cust2, hybrid's execution costs are similar to CSI, while having significant gains over B+ tree.

To better understand how the B+ tree and CSI are used in the same execution plan, Figure 10 provides a summary of statistics from the query plans chosen by the query optimizer in the presence of hybrid physical designs. The vertical bars report the percentage of leaf nodes in the plan which access columnstore (CSI) and B+ tree indexes respectively, averaged over all queries in the workload. The percentages are plotted on the primary vertical axis (left y-axis). The line reports the number of queries for which the optimizer chose an execution plan where both columnstore and B+ tree indexes were used. This number is plotted on the secondary vertical axis (right y-axis). As is evident from the figure, Cust1 and Cust3 leverage B+ tree indexes in most cases, though there are several plans where both types of indexes are used. On the other hand, Cust2's workload benefits more from columnstore, with a few hybrid plans.

Figure 10 provides strong evidence that for a variety of complex and real workloads, a hybrid physical design is beneficial. The benefits can vary depending on the workload characteristics and data distributions. A tuning tool, such as DTA, that can analyze and model the hybrid physical designs can help leverage the best of both columnstore and B+ tree indexes.

Note that DTA uses the query optimizer's estimated query plan costs to determine which combination of B+ tree and columnstore indexes is optimal for a given query. However, it is well-known that the query optimizer's estimates are not always accurate in terms of execution costs [26]. Such errors in the optimizer's estimate affect the recommendation quality of DTA, where in many cases it could result in sub-optimal recommendations. These are evident in Figure 9 for speedups less than 1 (and more noticeably, less than 0.5). In all cases, the hybrid physical design is still optimal in terms of optimizer's estimated cost. However, CSI and/or B+ tree plans are superior based on execution costs. As noted in Section 4.5, the hybrid physical designs require the optimizer to jointly estimate the costs of operators for vectorized (*batch mode*) and row-at-a-time (*row mode*) executions which adds an added layer of complexity, resulting in many more instances of cost estimate errors. SQL Server features such as Automatic Plan Correction [17] and adaptive operators [38] are useful to overcome such errors.

5.2.2 Mixed Workloads. We use the CH benchmark (scaling factor of 1000 warehouses) as a representative of mixed workloads common in operational databases also executing analytical workloads. The CH workload has two separate components: (a) threads executing the TPC-C transactions similar to the specification of the TPC-C benchmark; and (b) threads executing the H-like analysis queries. The C and H components share the same data. Since many queries execute concurrently, the queries contend for resources as well as locks. To minimize resource interference, we isolate the CPU cores for the two components, by affinitizing the C and H components to different sets of CPU cores using Resource Pool affinities in SQL Server [36]. In this experiment, we dedicate 30 cores for the H workload and the remaining 10 to the C workload. We use 20 client threads that generate the C and H workload in a tight loop without any think time, with 1 thread dedicated to H workload. We run the workload for six hours and use the median latency of each query/transaction type. Since columnstore and B+ tree indexes use different granularity and type of locking, the effect of lock contention in hybrid physical designs is also an aspect we report in our results. Therefore, we report the end-to-end wall clock time to execute the queries (instead of logical work done in CPU time) to better capture the concurrency effects. We also experiment with two different isolation levels, Snapshot Isolation (SI) and Serializable (SR) to observe this impact of locking.

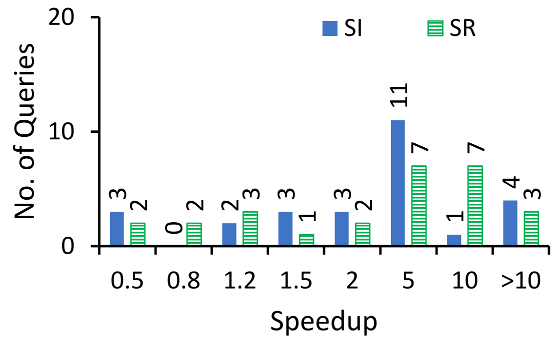


Figure 11: Distribution of speedup factor (median execution time) achieved by hybrid physical design compared to B+ tree-only for CH benchmark using Snapshot Isolation (SI) and Serializable (SR) isolation levels.

Figure 11 plots the distribution of speedup achieved by a hybrid physical design compared to a B+ tree-only design. Note that a columnstore-only design makes the C transactions extremely slow, thus slowing down all other queries due to lock contention. Hence, we consider two designs: B+ tree-only and hybrid. As expected, compared to a B+ tree-only design, a hybrid design significantly speeds up the H queries, while also resulting in moderate slowdown for some C transactions, primarily, the write transactions, NewOrder and Payment. Therefore, similar to our observation in Section 3, even for mixed workloads, a hybrid physical design allows us to leverage the best of B+ tree and columnstore. It is also interesting that using the Serializable isolation mode results in overall better latency improvements for the read-only queries, since Snapshot isolation creates multiple versions which makes reads slightly more expensive compared to Serializable which stores only a single version.

5.3 Example Hybrid Plans

We now drill into a few individual cases where hybrid physical design had at least an order of magnitude lower execution cost compared to columnstore-only. One such example is TPC-DS Query ID 54. This query references several large fact tables (e.g., `web_sales`, `store_sales`, etc.) as well as many dimension tables (e.g., `item`, `date_dim`, etc.). The query has several predicates on the dimensions, which are selective enough that B+ tree accesses are significantly cheaper than scanning the columnstore for the large fact tables. DTA recommends B+ tree indexes on several fact tables as well as a few dimensions where the selectivity is high, along with a few columnstore indexes on tables such as `customer_address` and `store`. In the presence of the B+ tree indexes, the optimizer uses index seek (using predicates on the dimensions) and nested loop joins to look up qualifying rows in the fact tables. On the other hand, with only columnstores, the optimizer scans the columnstore and uses hash joins. The CPU time spent on leaf nodes for the hybrid plan was about 25× lower than the leaf nodes in the columnstore-only plan. A similar pattern is also observed for Query ID 72, where in addition to B+ tree indexes on the fact tables, DTA also recommends B+ tree indexes on tables such as `household_demographics` and `customer_address` which are used in nested loops. There are several instances where a columnstore is built on a table in addition to a B+ tree and both indexes are referenced in the query plan.

We observed similar patterns in the real-world customer workloads. For instance, in the case of Cust4, there are several instances of DTA recommending a B+ tree index on the large fact table(s) and columnstore on the dimension tables. The optimizer uses an index seek on the fact table(s) followed by a scan of the columnstore on the dimensions, and joining the tables using hash join.

6 RELATED WORK

Over the past decade, many commercial DBMSs have added support for a columnstore, either as primary or secondary representation of data, as well as in-memory and on-disk [21–23, 25, 31–34, 39]. Some systems target columnstores primarily for data warehousing applications [33, 34, 39] while others have enabled them for general purpose DBMS applications [21, 25, 31] or for operational analytics (i.e., OLTP and decision support on the same database) [23, 32]. Our focus in this paper is the role of columnstore and B+ tree indexes on the same database supporting a variety of workloads, where the space of hybrid physical designs is important.

The need to select the appropriate set of access paths and physical designs has been an important problem even before the advent of columnstores. Several commercial systems have long supported physical design tuning tools that accompany their database engine. For instance, Database Engine Tuning Advisor for Microsoft SQL Server [4], DB2 Design Advisor for IBM DB2 [44], and SQL Access Advisor for Oracle [14, 35]. Similarly, for columnstores, Vertica supports Database Designer [22] that determines the sets of projections to build. Our extensions to DTA to support analyzing and recommending B+ tree and columnstore indexes in an integrated fashion is the first of its kind in a tuning tool. Previous approaches have also studied the impact of compression on physical design tuning [16, 20], though none of them study size estimation problems that arise due to a variety of encoding techniques used in CSIs.

Kester et al. [19] analyzed the role of access path selection in main-memory optimized data systems. While sharing the same goal to understand the performance trade-offs between CSI and B+ tree indexes, Kester et al. focused on a specific in-memory architecture that supports scan sharing and memory optimized B+ tree, considered one specific form of physical design (corresponding to our secondary B+ tree on top of CSI), and their primary focus was to study the problem in terms of concurrency. Moreover, the evaluation was using a prototype system. On the other hand, our analysis focuses on the analysis for a commercial-strength DBMS. Furthermore, our experimental analysis examines synthetic, mixed and several real-world customer workloads as well as a wide spectrum of physical designs. While Kester et al. proposed a model to estimate optimal concurrency among queries, our observations motivate the extensions to a commercial physical design tuning tool to analyze and recommend hybrid physical designs.

Abadi et al. [3] present an interesting experimental study quantifying the major differences between columnstore and row-oriented indexes such as B+ tree. The focus of that study was to understand the fundamental differences, and how one can be extended with properties of another. Arulraj et al. [5] explore a design where depending on the workload, the physical layout of the data automatically changes between row and column formats for different parts of the same table. SQL Server supports both B+ tree and columnstores on the same table and execution engine. Our focus in this study is to analyze how columnstore and B+ trees complement each other in hybrid physical designs.

Several systems, such as Hyper [18] and BatchDB [27], study the design and implementation of a DBMS to support a mix of OLTP and decision support workloads, similar to the mixed workload setup studied in this paper. These approaches rely on storage formats, sharing the data between the OLTP and the decision support components as well as isolating the workloads. We consider resource isolation between the OLTP and decision support workloads for our experiments with the CH workload in Section 5. However, our focus is on optimal choice of hybrid physical designs in a DBMS engine which supports both columnstore and B+ tree indexes.

7 CONCLUSIONS

Many commercial RDBMSs support columnstores and B+ tree indexes on the same database and table. We studied this design space of hybrid physical designs, where both columnstore and B+ tree indexes can be built on the same database and tables in the context of a commercial RDBMS. Our experimental analysis, using carefully-crafted micro-benchmarks, demonstrated that an appropriate combination of columnstore and B+ tree indexes can result in an order of magnitude better execution costs for several workload patterns. We presented an extension to Database Engine Tuning Advisor, a commercial-strength tuning tool for Microsoft SQL Server to analyze and automatically recommend a set of B+ tree and columnstore indexes appropriate for a specified workload. We conducted extensive experiments using a variety of industry-standard benchmarks as well as real-world customer workloads, which demonstrated that hybrid physical designs are indeed effective across many workloads. DTA can leverage the best of B+ tree and columnstores by using the workload to determine the appropriate columnstore-only, B+ tree-only, or hybrid recommendation. In the future, we plan to refine the columnstore size estimation algorithm and study other aspects that affect the execution costs for hybrid physical designs.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers, Craig Freedman, Vassilis Papadimos, and Miodrag Radulovic for their valuable feedback.

REFERENCES

[1] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases* 5, 3 (2013), 197–280.

[2] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating Compression and Execution in Column-oriented Database Systems. In *SIGMOD*. ACM, New York, NY, USA, 671–682. https://doi.org/10.1145/1142473.1142548

[3] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. 2008. Column-stores vs. Row-stores: How Different Are They Really?. In *SIGMOD*. ACM, New York, NY, USA, 967–980. https://doi.org/10.1145/1376616.1376712

[4] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. 2004. Database Tuning Advisor for Microsoft SQL Server 2005. In *VLDB*. Morgan Kaufmann, 1110–1121.

[5] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *SIGMOD*. 583–598. https://doi.org/10.1145/2882903.2915231

[6] Rudolf Bayer and Edward McCreight. 2002. Organization and maintenance of large ordered indexes. In *Software pioneers*. Springer, 245–262.

[7] Peter Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*.

[8] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. 1998. Random Sampling for Histogram Construction: How much is enough?. In *SIGMOD*. 436–447. https://doi.org/10.1145/276304.276343

[9] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*. 146–155. http://www.vldb.org/conf/1997/P146.PDF

[10] Surajit Chaudhuri and Vivek R. Narasayya. 1998. AutoAdmin 'What-if' Index Analysis Utility. In *SIGMOD*. 367–378. https://doi.org/10.1145/276304.276337

[11] Surajit Chaudhuri and Vivek R. Narasayya. 1999. Index Merging. In *ICDE*. 296–303. https://doi.org/10.1109/ICDE.1999.754945

[12] Richard L. Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi A. Kuno, Raghunath Othayoth Nambiar, Thomas Neumann, Meikel Pöss, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. 2011. The mixed workload CH-benCHmark. In *DBTest*.

[13] Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137.

[14] Benoît Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zaït, and Mohamed Ziauddin. 2004. Automatic SQL Tuning in Oracle 10g. In *VLDB*. 1098–1109. http://www.vldb.org/conf/2004/IND4P2.PDF

[15] Goetz Graefe. 2012. New algorithms for join and grouping operations. *Computer Science - Research and Development* 27, 1 (01 Feb 2012), 3–27. https://doi.org/10.1007/s00450-011-0186-9

[16] Stratos Idreos, Raghav Kaushik, Vivek R. Narasayya, and Ravishankar Ramamurthy. 2010. Estimating the compression fraction of an index using sampling. In *ICDE*. 441–444. https://doi.org/10.1109/ICDE.2010.5447871

[17] Jovan Popovic. 2017. Automatic plan correction in SQL Server 2017. https://blogs.msdn.microsoft.com/sqlserverstorageengine/2017/05/17/automatic-plan-correction-in-sql-server-2017/. (2017).

[18] Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*. IEEE Computer Society, Washington, DC, USA, 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[19] Michael S. Kester, Manos Athanassoulis, and Stratos Idreos. 2017. Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe?. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 715–730. https://doi.org/10.1145/3035918.3064049

[20] Hideaki Kimura, Vivek R. Narasayya, and Manoj Syamala. 2011. Compression Aware Physical Database Design. *Proc. VLDB Endow.* 4, 10 (2011), 657–668. http://www.vldb.org/pvldb/vol4/p657-kimura.pdf

[21] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, Juan Loaiza, Neil MacNaughton, Vineet Marwah, Niloy Mukherjee, Atrayee Mullick, Sujatha Muthulingam, Vivekanandhan Raja, Marty Roth, Ekrem Soylemez, and Mohamed Zaït. 2015. Oracle Database In-Memory: A dual format in-memory database. In *ICDE*. 1253–1258. https://doi.org/10.1109/ICDE.2015.7113373

[22] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *Proc. VLDB Endow.* 5, 12 (2012), 1790–1801.

[23] Per-Åke Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michał Nowakiewicz, and Vassilis Papadimos. 2015. Real-time Analytical Processing with SQL Server. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1740–1751. https://doi.org/10.14778/2824032.2824071

[24] Per-Åke Larson, Cipri Clinciu, Campbell Fraser, Eric N. Hanson, Mostafa Mokhtar, Michał Nowakiewicz, Vassilis Papadimos, Susan L. Price, Srikumar Rangarajan, Remus Rusanu, and Mayukh Saubhasik. 2013. Enhancements to SQL Server column stores. In *SIGMOD*.

[25] Per-Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. 2011. SQL Server Column Store Indexes. In *SIGMOD*. ACM, New York, NY, USA, 1177–1184. https://doi.org/10.1145/1989323.1989448

[26] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (November 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[27] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. 2017. BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications. In *SIGMOD*. 37–50. https://doi.org/10.1145/3035918.3035959

[28] Microsoft. 2016. SQL Server column-store. https://docs.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-overview. (2016).

[29] Microsoft SQL Server. 2017. Query Store. https://docs.microsoft.com/en-us/sql/relational-databases/performance/monitor-and-tune-for-performance. (2017).

[30] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 476–487. http://dl.acm.org/citation.cfm?id=645924.671173

[31] Oracle White Paper 2018. Oracle Database In-Memory with Oracle Database 18c. http://www.oracle.com/technetwork/database/in-memory/overview/twp-oracle-database-in-memory-2245633.pdf. (February 2018).

[32] Hasso Plattner. 2009. A common database approach for OLTP and OLAP using an in-memory column database. In *SIGMOD*. 1–2. https://doi.org/10.1145/1559845.1559846

[33] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *Proc. VLDB Endow.* 6, 11 (2013), 1080–1091.

[34] Dominik Ślęzak, Jakub Wróblewski, Victoria Eastwood, and Piotr Synak. 2008. Brighthouse: An Analytic Data Warehouse for Ad-hoc Queries. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1337–1345. https://doi.org/10.14778/1454159.1454174

[35] SQL Access Advisor 2017. SQL Access Advisor. https://docs.oracle.com/cd/B19306_01/server.102/b14211/advisor.htm#i1008370. (2017).

[36] sql server resource pools 2017. Resource Governor Resource Pool. https://docs.microsoft.com/en-us/sql/relational-databases/resource-governor/resource-governor-resource-pool. (2017).

[37] SQL Server Team. 2017. Announcing Columnstore Indexes and Query Store support in Database Engine Tuning Advisor. https://blogs.technet.microsoft.com/dataplatforminsider/2017/01/10/announcing-columnstore-indexes-and-query-store-support-in-database-engine-tuning-advisor/. (2017).

[38] SQL Server Team. 2017. Enhancing query performance with Adaptive Query Processing in SQL Server 2017. https://blogs.technet.microsoft.com/dataplatforminsider/2017/09/28/enhancing-query-performance-with-adaptive-query-processing-in-sql-server-2017/. (2017).

[39] Mike Stonebraker, Daniel Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-store: a column-oriented DBMS. In *VLDB*.

[40] The Transaction Processing Council. 2017. TPC Benchmark DS: Standard Specification v2.6.0. http://www.tpc.org/tpcds/. (2017).

[41] The Transaction Processing Council. 2017. TPC-C Benchmark: Standard Specification. http://www.tpc.org/tpcc/. (2017).

[42] The Transaction Processing Council. 2017. TPC-H Benchmark: Standard Specification. http://www.tpc.org/tpch/. (2017).

[43] Vertica Analytics Platform. 2017. Projection Types in Vertica Analytics Platform 8.1.x. https://my.vertica.com/docs/8.1.x/HTML/index.htm#Authoring/ConceptsGuide/Components/ProjectionContentTypes.htm. (2017).

[44] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam J. Storm, Christian Garcia-Arellano, and Scott Fadden. 2004. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB*. 1087–1097. http://www.vldb.org/conf/2004/IND4P1.PDF

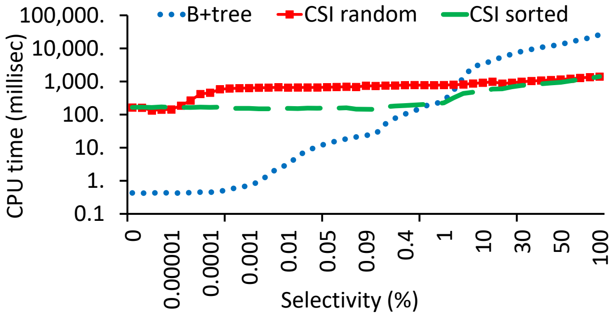


Figure 12: CPU time for B+ tree and CSI (sorted and unsorted) for a query with varying selectivity.

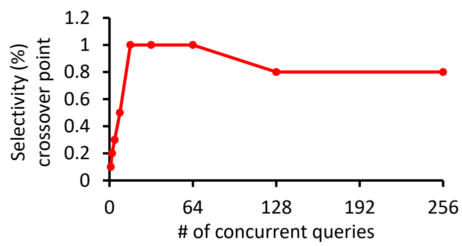


Figure 13: The selectivity (%) crossover point between B+ tree and CSI for concurrent queries.

A DETAILS ON EVALUATION OF HYBRID PHYSICAL DESIGNS

In this section, we present extended and more detailed results of the experiments described in section 3.

A.1 Fully sorted columnstores

In Section 3.2.1, we empirically quantified how sortedness in columnstores can be leveraged for aggressive data skipping that helps execution time. Figure 2(a) reported the crossover point in terms of end-to-end query execution time. In Figure 12, we plot the CPU time for the queries to analyze the crossover point in terms of the work done by the query. As noted in Figure 2(b), when accessing more than about 4% of data, the efficient parallel scan of CSI outperforms the parallel scan of B+ trees. However, such parallel plans using multiple threads consume more CPU resources compared to the serial plan used for the B+ tree for small selectivity values. This aspect is evident in Figure 12 where the crossover point for a sorted CSI and B+ tree is at a much larger selectivity compared to that in terms of execution time reported in Figure 2(a). The biggest advantage of the B+ trees is when the selective fetching of pages from disk saves I/O and CPU time as less data has to be processed with an efficient single-threaded execution plan. A sorted columnstore ameliorates some of these overheads by allowing more data skipping. However, since the amount of data processed with a sorted CSI is still much larger than that of the B+ tree, the optimizer still chooses a multi-threaded plan which has higher CPU requirements.

A.2 Concurrent queries

As is evident from the experiment in Appendix A.1, since B+ trees skip more data for queries with low selectivity values, they can run single-threaded and consume less CPU. Such CPU-efficiency allows for more queries to execute concurrently, thus resulting in lower execution time for concurrent queries. To quantify this effect, we repeat the experiment from Section 3.2.1 by varying the number of concurrent queries executing. We use the same query Q_1 executing on data cached in memory (i.e., hot runs) and vary the number of threads concurrently issuing the same query. In Figure 13, we present how the selectivity (%) crossover point (for end-to-end query execution time) changes depending on the number of concurrent queries. We vary the number of concurrent queries from 1 to 256 in multiples of 2. Each data point in Figure 13 corresponds to plotting the graph as in Figure 1(a) with a given number of concurrent queries and reporting its crossover point. This experiment is modeled similar to Kester et al. [19] (see Sections 2.4 and 4) to quantify the impact on SQL Server which has a traditional B+ tree designed for disk-based systems and does not support scan sharing. We only report the crossover point with end-to-end execution time since the crossover point for CPU time is not affected much by concurrent queries. The crossover point is similar to that reported in Figure 1(b).

The B+ tree index uses a single thread for low selectivity queries and even after the switch to multi-threaded execution at the selectivity value of about 0.2%, it processes less data and thus its CPU time becomes on par with CSI only after the selectivity value of about 1%. For small number of concurrent queries (≤ 8), there is enough available CPU on the server such that there is little CPU contention for the resource-intensive parallel scans on columnstore. Therefore, in such cases, the B+ tree is beneficial only for very small selectivity values as the serial scans do not fully leverage the free resources. However, as the number of concurrent queries increase, the scans with columnstore encounters contention and blocking for CPU, which increases the end-to-end execution time, thus shifting the crossover point higher. However, beyond a certain degree of concurrency, even the serial plans on B+ tree starts contending for CPU. In such cases, since the CSI is more CPU-efficient per unit data processed, the crossover point is lower. Note that SQL Server does not support shared scans which would make scans in this experimental setup more efficient, as observed in Kester et al. [19].