Pushing Data-Induced Predicates Through Joins in Big-Data Clusters

Srikanth Kandula, Laurel Orr, Surajit Chaudhuri
Microsoft

ABSTRACT

Using data statistics, we convert predicates on a table into data induced predicates (diPs) that apply on the joining tables. Doing so substantially speeds up multi-relation queries because the benefits of predicate pushdown can now apply beyond just the tables that have predicates. We use diPs to skip data exclusively during query optimization; i.e., diPs lead to better plans and have no overhead during query execution. We study how to apply diPs for complex query expressions and how the usefulness of diPs varies with the data statistics used to construct diPs and the data distributions. Our results show that building diPs using zone-maps which are already maintained in today's clusters leads to sizable data skipping gains. Using a new (slightly larger) statistic, 50% of the queries in the TPC-H, TPC-DS and JoinOrder benchmarks can skip at least 33% of the query input. Consequently, the median query in a production big-data cluster finishes roughly 2× faster.

PVLDB Reference Format:
Srikanth Kandula, Laurel Orr, Surajit Chaudhuri. Pushing Data-Induced Predicates Through Joins in Big-Data Clusters. *PVLDB*, 12(3): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/3368289.3368292

1. INTRODUCTION

In this paper, we seek to extend the benefits of predicate pushdown beyond just the tables that have predicates. Consider the following fragment of TPC-H query #17 [20].

```
SELECT SUM(l_extendedprice)
FROM lineitem
JOIN part ON l_partkey = p_partkey
WHERE p_brand=':1' AND p_container=':2'
```

The `lineitem` table is much larger than the `part` table, but because the query predicate uses columns that are only available in `part`, predicate pushdown cannot speed up the scan of `lineitem`. However, it is easy to see that scanning the entire `lineitem` table will be wasteful if only a small number of those rows will join with the rows from `part` that satisfy the predicate on `part`.

If only the predicate was on the column used in the join condition, `_partkey`, then a variety of techniques become applicable (e.g., algebraic equivalence [53], magic set rewriting [50, 72] or value-based pruning [81]), but predicates over join columns are rare,[1] and these techniques do not apply when the predicates use columns that do not exist in the joining tables.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 12, No. 3
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3368289.3368292

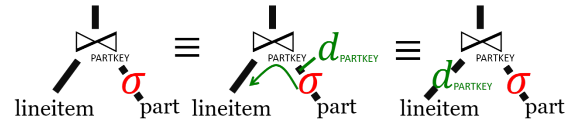


Figure 1: Example illustrating creation and use of a data-induced predicate which only uses the join columns and is a necessary condition to the true predicate, i.e., $\sigma \Rightarrow d_{\texttt{partkey}}$.

Some systems implement a form of sideways information passing over joins [21, 68] during query execution. For example, they may build a bloom filter over the values of the join column `_partkey` in the rows that satisfy the predicate on the `part` table and use this bloom filter to skip rows from the `lineitem` table. Unfortunately, this technique only applies during query execution, does not easily extend to general joins and has high overheads, especially during parallel execution on large datasets because constructing the bloom filter becomes a scheduling barrier delaying the scan of `lineitem` until the bloom filter has been constructed.

We seek a method that can convert predicates on a table to data skipping opportunities on joining tables even if the predicate columns are absent in other tables. Moreover, we seek a method that applies exclusively during query plan generation in order to limit overheads during query execution. Finally, we are interested in a method that is easy to maintain, applies to a broad class of queries and makes minimalist assumptions.

Our target scenario is big-data systems, e.g., SCOPE [45], Spark[37, 85], Hive [80], F1 [76] or Pig [67] clusters that run SQL-like queries over large datasets; recent reports estimate over a million servers in such clusters [1].

Big-data systems already maintain data statistics such as the maximum and minimum value of each column at different granularities of the input; details are in Table 1. In the rest of this paper, for simplicity, we will call this the zone-map statistic and we use the word partition to denote the granularity at which statistics are maintained.

Using data statistics, we offer a method that converts predicates on a table to data skipping opportunities on the joining tables at query optimization time. The method, an example of which is shown in Figure 1, begins by using data statistics to eliminate partitions on tables that have predicates. This step is already implemented in some systems [7, 18, 45, 81]. Next, using the data statistics of the partitions that satisfy the local predicates, we construct a new predicate which

[1]Over all the queries in TPC-H [26] and TPC-DS [24], there are zero predicates on join columns perhaps because join columns tend to be opaque system-generated identifiers.

Table 1: Data statistics maintained by several systems.

Scheme	Statistic	Granularity
ZoneMaps [14]	max and min value per column	*zone*
Spark [9, 18]		*file*
Exadata [64]	max, min and null present or null count per column	per table *region*
Vertica [59], ORC [2], Parquet [16]		stripe, row-group
Brighthouse [77]	histograms, char maps per col	*data pack*

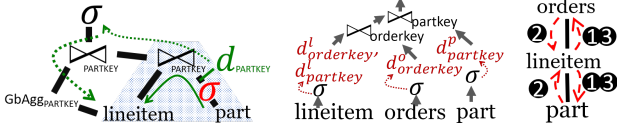


Figure 2: Illustrating the need to move diPs past other operations (left). On a 3-way join when all tables have predicates (middle), the optimal schedule only requires three (parallel) steps (right).

captures all of the join column values contained in such partitions. This new *data-induced predicate* (diP) is a necessary condition of the actual predicate (i.e., $\sigma \Rightarrow d$) because there may be false-positives; i.e., in the partitions that are included in the diP, not all of the rows may satisfy σ. However, the diP can apply over the joining table because it only uses the join column; in the case of Figure 1, the diP constructed on the `part` table can be applied on the partition statistics of `lineitem` to eliminate partitions. All of these steps happen during query optimization; our QO effectively replaces each table with a partition subset of that table; the reduction in input size often triggers other plan changes (e.g., using broadcast joins which eliminate a partition-shuffle [47]) leading to more efficient query plans.

If the above method is implemented using zone-maps, which are maintained by many systems already, the only overhead is an increase in query optimization time which we show is small in §5.

For queries with joins, we show that data-induced predicates offer comparable query performance at much lower cost relative to materializing denormalized join views [46] or using join indexes [5, 23]. The fundamental reason is that these techniques use augmentary data-structures which are costly to maintain; yet, their benefits are limited to narrow classes of queries (e.g., queries that match views, have specific join conditions, or very selective predicates) [34]. Data-induced predicates, we will show, are useful more broadly.

We also note that the construction and use of data-induced predicates is decoupled from how the datasets are laid out. Prior work identifies useful data layouts, for example, co-partition tables on their join columns [51, 62] or cluster rows that satisfy the same predicates [73, 78]; the former speeds up joins and the latter enhances data skipping. In our big-data clusters, many unstructured datasets remain in the order that they were uploaded to the cluster. The choice of data layout can have exogenous constraints (e.g., privacy) and is query dependent; that is, no one layout helps with all possible queries. In §5, we we will show that diPs offer significant additional speedup when used with the data layouts proposed by prior works and that diPs improve query performance in other layouts as well.

To the best of our knowledge, this paper is the first to offer data skipping gains across joins for complex queries before query execution while using only small per-table statistics. Prior work either only offers gains during query execution [21, 50, 53, 68, 72, 81] or uses more complex structures which have sizable maintenance overheads [5, 23, 46, 73, 78]. To achieve the above, we offer an efficient method to compute diPs on complex query expressions (multiple joins, join cycles, nested statements, other operations). This method works with a variety of data statistics. We also offer a new statistic,

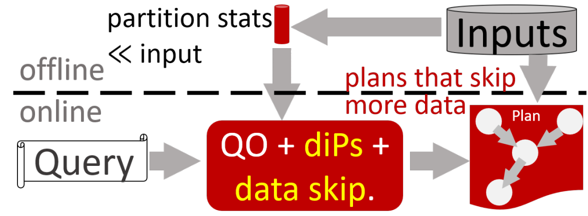


Figure 3: A workflow which shows changes in red; using partition statistics, our query optimizer computes data-induced predicates and outputs plans that read less input.

range-set, that improves query performance over zone-maps at the cost of a small increase in space. We also discuss how to maintain statistics when datasets evolve. In more detail, the rest of this paper has these contributions.

- Using diPs for complex queries leads to novel challenges.
  - Consider TPC-H query#17 [20] in Figure 2(left) which has a nested sub-query on the `lineitem` table. Creating diPs for only the fragment considered in Figure 1 still reads the entire `lineitem` table for the nested group-by. To alleviate this, we use new QO transformation rules to move diPs; in this case, shown with dotted arrows, the diP is pulled above a join, moved sideways to a different join input and then pushed below a group-by thereby ensuring that a shared scan of `lineitem` will suffice.
  - When multiple joining tables have predicates, a second challenge arises. Consider the 3-way join in Figure 2(middle) where all tables have local predicates. The figure shows four diPs: one per table and per join condition. If applying these diPs eliminates partitions on some joining table, then the diPs that were previously constructed on that table are no longer up-to-date. Re-creating diPs whenever partition subsets change will increase data skipping but doing so naïvely can construct excessively many diPs which increases query optimization time. We present an optimal schedule for tree-like join graphs which converges to fixed point and hence achieves all possible data skipping while computing the fewest number of diPs. Star and snowflake schemas result in tree-like join graphs. We discuss how to derive diPs for general join graphs within a cost-based query optimizer in §3.
- We show how different data statistics can be used to compute diPs in §4 and discuss why *range-sets* represent a good trade-off between space usage and potential for data skipping.
- We discuss two methods to cope with dataset updates in §4.1. The first method *taints* partitions whose rows change and the second *approximately updates* partition statistics when rows change. We will show that both methods can be implemented efficiently and that diPs offer large gains in spite of updates.
- Fundamentally, data-induced predicates are beneficial only if the join column values in the partitions that satisfy a predicate contain only a small portion of all possible join column values. In §2.1, we discuss real-world use-cases that cause this property to hold and quantify their occurrence in production workloads.
- We report results from experiments on production clusters at Microsoft that have tens of thousands of servers. We also report results on SQL server. See Figure 3 for a high-level architecture diagram. Our results in §5 will show that using small statistics and a small increase in query optimization time, diPs offer sizable gains on three workloads (TPC-H [26], TPC-DS [24], JOB [12]) under a variety of conditions.

2. MOTIVATION

We begin with an example that illustrates how data-induced predicates (diPs) can enhance data skipping. Consider the query expression, $\sigma_{\text{year}}(\texttt{date_dim}) \bowtie_{\texttt{date_sk}} R$. Table 2a shows the zone-maps per partition for the predicate and join columns. Recall that zone-maps are the maximum and minimum value of a column in each partition, and we use partition to denote the granularity at which statistics are maintained which could be a file, a rowgroup etc. (see Table 1). Table 2b shows the diPs corresponding to different predicates. The predicate column `year` is only available on the `date_dim` table, but the diPs are on the join column `date_sk` and can be pushed onto joining relations using column equivalence [53]. The diPs shown here are DNFs over ranges; if the equijoin condition has multiple columns, the diPs will be a conjunction of DNFs, one DNF per column. Further details on the class of predicates supported, extending to multiple joins and handling other operators, are in §3.2. Table 2b also shows that the diPs contain a small portion of the range of the join column `date_sk` (which is [1000, 12000]); thus, they can offer large data skipping gains on joining relations.

It is easy to see that diPs can be constructed using any data statistic that supports the following steps: (1) identify partitions that satisfy query predicates, (2) *merge* the data statistic of the join columns over the satisfying partitions, and (3) use the merged statistic to extract a new predicate that can identify partitions satisfying the predicate in joining relations. Many data statistics support these steps [31], and different stats can be used for different steps.

To illustrate the trade-offs in choice of data statistics, consider Figure 4a which shows equi-width histograms for the same columns and partitions as in Table 2a. A histogram with b buckets uses $b+2$ doubles[2] compared to the two doubles used by zone maps (for the min. and max. value). Regardless of the number of buckets used, note that histograms will generate the same diPs as zone-maps. This is because histograms do not remember *gaps* between buckets. Other histograms (e.g., equi-depth, v-optimal) behave similarly. Moreover, the frequency information maintained by histograms is not useful here because diPs only reason about the existence of values. Guided by this intuition, consider a set of non-overlapping ranges $\{[l_i, u_i]\}$ which contain all of the data values; such *range-sets* are a simple extension of zone-maps which are, trivially, range-sets of size 1. However, range-sets also record gaps that have no values. Figure 4b shows range-sets of size 2. It is easy to see that range-sets give rise to more succinct diPs[3]. We will show that using a small number of ranges leads to sizable improvements to query performance in §5. We discuss how to maintain range-sets and why range-sets perform better than other statistics (e.g., bloom filters) in §4.

To assess the overall value of diPs, for TPC-H query #17 [20] from Figure 2(left), Figure 5 shows the I/O size reduction from using diPs. These results use a range-set of size 4 (i.e., 8 doubles per column per partition). The TPC-H dataset was generated with a scale factor of 100, skewed with a zipf factor of 2 [29], and tables were laid out in a typical manner[4]. Each partition is ~ 100MBs of data which is a typical quanta in distributed file systems [40] and is the default in our clusters [88]. Recall that the predicate columns are only available in the `part` table. The figure shows that only two partitions of `part` contain rows that satisfy the predicate and the corresponding diP eliminates many partitions in `lineitem`. We will show results in §5 for many different data layouts and data distributions. We discuss plan transformations needed to move the diP, as shown in Figure 2 (left), in §3.3. Overall, for the 100GB dataset, a 0.5MB statistic reduces the initial I/O for this query by 20×; the query can speed up by more or less depending on the work remaining after initial I/O.

[2] b to store the frequency per bucket and two for min and max.

[3] For `year` ≤ 1995, the diP using two ranges is `date_sk` ∈ $\{[1K, 2K], [3K, 3.5K], [4K, 6K]\}$ which covers 30% fewer values than the diP built using a zone-map (`date_sk` ∈ $[1K, 6K]$) in Table 2b.

[4] `lineitem` was clustered on `l_shipdate` and each cluster sorted on `l_orderkey`; `part` was sorted on its key; this layout is known to lead to good performance because it reduces re-partitioning for joins and allows date predicates to skip partitions [3, 22, 27].

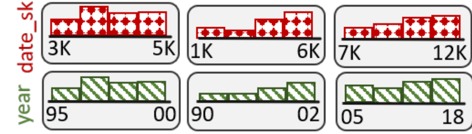
Table 2: Constructing diPs using partition statistics.

Column	Partition # 1	2	3
date_sk	[3000, 5000]	[1000, 6000]	[7000, 12000]
year	[1995, 2000]	[1990, 2002]	[2005, 2018]

(a) Zone maps [14], i.e., the maximum and minimum values, for two columns in three hypothetical partitions of the `date_dim` table.

Pred. (σ)	Satisfying partitions	Data-induced Predicate	% total range
year ≤ 1995	{1, 2}	date_sk ∈ [1000, 6000]	45%
year ∈ [2003, 2004]	∅	date_sk ∈ []	0%
year > 2010	{3}	date_sk ∈ [7000, 12000]	45%

(b) Data-induced predicates on the join column `date_sk` corresponding to predicates on the column `year`; built using stats from Table 2a.



(a) Equiwidth histograms for the dataset in Table 2a.

Range-set (size 2)	Partition # 1	2	3
date_sk	{[3000, 3500], [4000, 5000]}	{[1000, 2000], [5000, 6000]}	{[7000, 10000], [11000, 12000]}
year	{[1995, 1997], [1998, 2000]}	{[1990, 1993], [1998, 2002]}	{[2005, 2014], [2015, 2018]}

(b) Range-set of size 2, i.e., two non-overlapping max and min values, which contain all of the data values.

Figure 4: Other data statistics (histograms, range-sets) for the same example as in Table 2a; range-sets yield more succinct diPs.

#Partitions remaining in...

Step	lineitem	part
Initial	1000	26
After predicate	1000	2
diP: part → lineitem	50	2

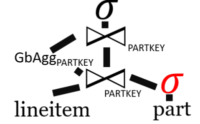


Figure 5: For TPC-H query 17 in Figure 2 (left), the table shows the partition reduction from using diPs. On the right, we show the plan generated using magic-set transformations which push group-by above the join. diPs complement magic-set transformations; we see here that magic-set tx cannot skip partitions of `lineitem` but because group-by has been pushed above the join, moving diPs sideways once is enough unlike the case in Figure 2(left).

2.1 Use-cases where data-induced predicates can lead to large I/O savings

Given the examples thus far, it is perhaps easy to see that diPs translate into large I/O savings when the following conditions hold.

- C1 The predicate on a table is satisfied by rows belonging to a small subset of partitions of that table.
- C2 The join column values in partitions that satisfy the predicate are a small subset of all possible join column values.

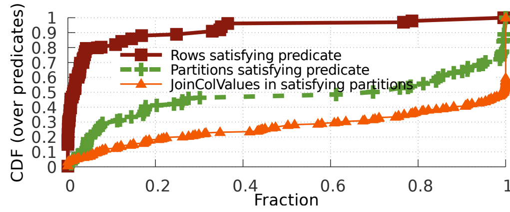


Figure 6: Quantifying how often the conditions that lead to large I/O skipping gains from using diPs hold in practice by using queries and datasets from production clusters at Microsoft.

C3 In tables that receive diPs, the join column values are distributed such that diPs only pick a small subset of partitions.

We identify use-cases where these conditions hold based on our experiences in production clusters at Microsoft [45].

- Much of the data in production clusters is stored in the order in which it was ingested into the cluster [37, 41]. A typical ingestion process consists of many servers uploading data in large batches. Hence, a consecutive portion of a dataset is likely to contain records for roughly similar periods of time, and entries from a server are concentrated into just a few portions of the dataset. Thus, queries for a certain time-period or for entries from a server will pick only a few portions of the dataset. This helps with C1. When such datasets are joined on time or server-id, this phenomenon also helps with C2 and C3.
- A common physical design methodology for performant parallel plans is to hash partition a table by predicate columns and range partition or order by the join columns [3, 22, 27, 51] and vice-versa. Performance improves since the shuffles to re-partition for joins decrease [17, 47, 89] and predicates can skip data. Such data layouts help with all three conditions C1–C3 and, in our experiments, receive the largest I/O savings from diPs.
- Join columns are keys which monotonically increase as new data is inserted and hence are related to time. For example, both the title-id of movies and the name-id of actors in the IMDB dataset [11] roughly monotonically increase as each new title and new actor are added to the dataset. In such datasets, predicates on time as well as predicates that are implicitly related to time, such as co-stars, will select only a small range of join column values. This helps with C1 and C2.
- Practical datasets are skewed; often times the skew is heavy-tailed [32]. In skewed datasets, predicates and diPs that skip over the heavy hitters are highly selective; hence, skew can help C1–C3.

Figure 6 illustrates how often conditions C1 and C2 hold for different datasets, query predicates and join columns from production clusters at Microsoft. We used tens of datasets and extracted predicates and join columns from thousands of queries. The figure shows the cumulative distribution functions (CDFs) of the fraction of rows satisfying each predicate (red squares), the fraction of partitions containing these rows (green pluses) and the fraction of join column values contained in these partitions (orange triangles). We see that about 40% of the predicates pick less than 20% of partitions (C1)[5]; in about 30% of the predicates, the join column values contained in the partitions satisfying the predicate are less than 50% of all join column values (C2).

3. CONSTRUCTION AND USE OF DATA-INDUCED PREDICATES

We describe our algorithm to enhance data skipping using data-induced predicates. Given a query $\mathcal{E}$ over some input tables, our goal is to emit an equivalent expression $\mathcal{E}'$ in which one or more of the table accesses are restricted to only read a subset of partitions. The algorithm applies to a wide class of queries (see §3.2) and can work with many kinds of data statistics (see §4).

[5]Read the value of green pluses line at $x = 0.2$ in Figure 6.

Table 3: Notation used in this paper.

Symbol	Meaning
p_i	Predicate on table i
p_{ij}	Equi-join condition between tables i and j
q_i	A vector whose x'th element is 1 if partition x of table i has to be read and 0 otherwise.
$d_{i\to j}$	Data-induced predicate from table i to table j
partition	granularity at which the store maintains statistics (Table 1)

The algorithm has three building blocks: use predicates on individual tables to identify satisfying partitions, construct diPs for pairs of joining tables and apply diPs to further restrict the subset of partitions that have to be read on each table. Using the notation in Table 3, these steps can be written as:

$$\forall \text{ table } i,\ \text{partition } x, \qquad q_i^x \leftarrow \mathsf{Satisfy}(p_i, x), \tag{1}$$

$$\forall \text{ tables } i, j, \qquad d_{i\to j} \leftarrow \mathsf{DataPred}(q_i, p_{ij}), \tag{2}$$

$$\forall \text{ table } j,\ \text{partition } x, \quad q_j^x \leftarrow q_j^x \prod_{i\neq j} \mathsf{Satisfy}(d_{i\to j}, x). \tag{3}$$

We defer describing how to efficiently implement these equations to §4 because the details vary based on the statistic and focus here on using these building blocks to enhance data skipping.

Note that the first step (Equation 1) executes once, but the latter two steps may execute multiple times because whenever an incoming diP changes the set of partitions that have to be read on a table (i.e., q changes in Equation 3), then the diPs from that table (which are computed in Equation 2 based on q) will have to be recomputed. This effect may cascade to other tables.

If a *join graph*, constructed with tables as nodes and edges between tables that have a join condition, has n nodes and m edges, then a naïve method will construct $2m$ diPs using Eq. 2, one along each edge in each direction, and will use these diPs in Eq. 3 to further restrict the partition subsets of joining tables. This step repeats until fixpoint is reached (i.e., no more partitions can be eliminated). Acyclic join graphs can repeat this step up to $n-1$ times, i.e., construct up to $2m(n-1)$ diPs, and join graphs with cycles can take even longer (see [19] for an example). Abandoning this process before the partition subsets converge can leave data skipping gains untapped. On the other hand, generating too many diPs adds to query optimization time. To address this challenge, we construct diPs in a carefully chosen order so as to converge to the smallest partition subsets while building the minimum number of diPs (see §3.4).

A second challenge arises when applying the above method, which only accounts for select and join operations, to the general case where queries contain many other interceding operations such as group-bys and nested statements. One option is to ignore other operations and apply diPs only to sub-portions of the query that exclusively consist of selections and joins. Doing so, again, leaves data skipping gains untapped; in some cases the unrealized gains can be substantial as we saw for the query in Figure 2 (left) where ignoring the nested statement (that is, restricting diPs to just the portion shown with a shaded background in the figure) may lead to no gains since the group-by can require reading the `lineitem` table fully. To address this challenge, we move diPs around other relational operators using commutativity. We list the transformation rules used in §3.3 which cover a broad class of operators. Using these transformations extends the usefulness of diPs to complex query expressions.

3.1 Deriving diPs within a cost-based QO

Taken together, the previous paragraphs indicate two requirements to quickly identify efficient plans: (1) carefully schedule the order in

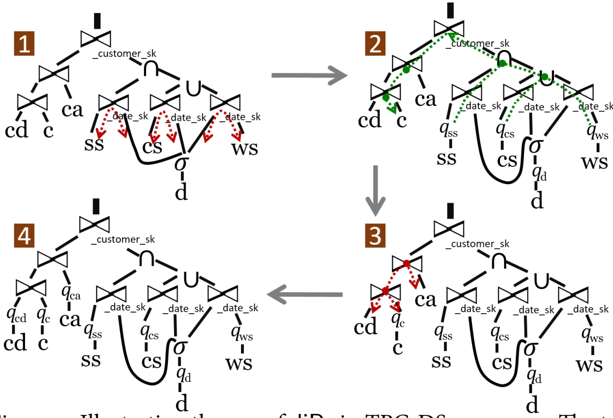


Figure 7: Illustrating the use of diPs in TPC-DS query #35. The table labels `ss`, `cs`, `ws` and `d` correspond to the tables `store_sales`, `catalog_sales`, `web_sales` and `date_dim`.

which diPs are computed over a join graph and (2) use commutativity to move diPs past other operators in complex queries. We sketch our method to derive diPs within a cost-based QO here.

Let's consider some alternative designs. (1) Could the user or a query rewriting software that is separate from the QO insert optimal diPs into the query? This option is problematic because the user or the query rewriter will have to re-implement complex logic such as predicate simplification and push-down that is already available within the QO. Furthermore, moving diPs around other operators (see §3.3) requires plan transformation rules that are not implemented in today's QO; specifically rules that *pull up* diPs or *move them sideways* from one join input to another do not exist in typical QOs. As we saw with the case of the example in Figure 2(left), without such movement diPs may not achieve any data skipping. (2) Could the change to QO be limited to adding some new plan transformation rules? Doing so is appealing since the QO framework remains unchanged. Unfortunately, as we saw in the case of Figure 2(middle), diPs may have to be exchanged multiple times between the same pair of tables, and to keep costs manageable, diPs have to be constructed in a careful order over the join graphs; in today's cost-based optimizers, achieving such recursion and fine-grained query-wide ordering is challenging [53]. Thus, we use the hybrid design discussed next.

We add derivation of diPs as a new phase in the QO after plan simplification rules have applied but before exploration, implementation, and costing rules, such as join ordering and choice of join implementations, are applied. The input to this phase is a logical expression where predicates have been simplified and pushed down. The output is an equivalent expression which replaces one or more tables with partition subsets of those tables. To speed up optimization, this phase creates maximal sub-portions of the query that only contain selections and joins; we do this by pulling up group-bys, projections, predicates that use columns from multiple relations, etc. diPs are exchanged within these maximal select-join sub-portions of the query expression using the schedule in §3.4. Next, using the rules in §3.3, diPs are moved into the rest of the query. With this method, derivation will be faster when the select-join sub-portions are large because, by decoupling the above steps, we avoid propagating diPs which have not converged to other parts of the query. Note that this phase executes exactly once for a query. The increase in query optimization time is small, and by exploring alternative plans later, the QO can find plans that benefit from the reduced input sizes (e.g., choose a different join order or use broadcast join instead of hashjoin).

Example: Figure 7 illustrates this process for TPC-DS query #35; the SQL query is in [25]. As shown in the top left of the figure, labeled 1, diPs that are triggered by the predicate on `date_dim` are first exchanged in maximal SJ portions: `store_sales` ⋈ `date_dim`, `catalog_sales` ⋈ `date_dim` and `web_sales` ⋈ `date_dim`. The query joins these portions with another join expression after a few set operations. Hence, in 2, we build new diPs for the `customer_sk` column and pull those up through the set operations (union and intersection translate to logical or and and over diPs) and push down to the `customer` (c) table. To do so, we use the transformation rules in §3.3. In 3, if the incoming diP skips partitions on the `customer` table, another derivation ensues within the SJ expression on the left.[6] The final plan, shown in 4, effectively replaces each table with the partition subset that has to be read from that table.

3.2 Supported Queries

Our prototype does not restrict the query class, i.e., queries can use any operator supported by the underlying platform. Here, we highlight aspects that impact the construction and use of diPs.

Predicates: Our prototype triggers diPs for predicates which are conjunctions, disjunctions or negations over the following clauses:

- c op v: here c is a numeric column, op denotes an operation that is either $=, <, \leq, >, \geq, \neq$ and v is a value.
- c_i op c_j: here c_i, c_j are numeric columns from the same relation and op is either $=, <, \leq, >, \geq, \neq$.
- For string and categorical columns, equality check with a value.

Joins: Our prototype generates diPs for join conditions that are column equality over one or more columns; although, extending to some other conditions (e.g., band joins [4]) is straightforward. We support inner, one-sided outer, semi, anti-semi and self joins.

Projections: diPs commute trivially with any projection on columns that are not used in the diP. On columns that are used in a diP, only single-column invertible projections commute with that diP because only such projects can be inverted through zone-maps and other data statistics that we use to compute diPs (see [19]).

Other operations: Operators that do not commute with diPs will block the movement of diPs. As we discuss in §3.3 next, diPs commute with a large class of operations.

3.3 Commutativity of data-induced predicates with other operations

We list some query optimizer transformation rules that apply to data-induced predicates (diPs). The correctness of these rules follows from considering a diP as a filter on join columns. Note that some of these transformation are not used in today's query optimizers. For example, *pulling up* diPs above a union and a join (rule #4, #5, below) naively results in redundant evaluation of predicates and are hence not used today; however, as we saw in the case of Figure 7, such movements are necessary to skip partitions elsewhere in the query. We also note that diPs do not remain in the query plan; the diPs directly on tables are replaced with a read of the partition subsets of that table, and other diPs are dropped.

1. diPs commute with any select.
2. A diP commutes with any projection that does not affect the columns used in that diP. For projections that affect columns used in a diP, commutativity holds if and only if the projections are invertible functions on one column.

[6] After 3, if the partition subset on the `customer` table becomes further restricted, a new diP moves in opposite direction along the path shown in 2; we do not discuss this issue for simplicity.

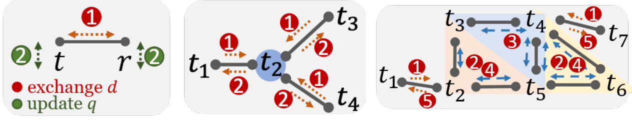


Figure 8: Schedules of exchanging diPs for different join graphs; numbers-in-circles denote the epoch; multiple diPs are exchanged in parallel in each epoch. Details are in §3.4.

3. diPs commute with a group-by if and only if the columns used in the diP are a subset of the group-by columns.
4. diPs commute with set operations such as union, intersection, semi- and anti semi-joins, as shown below.
   - $d_1(\mathcal{R}_1) \cap d_2(\mathcal{R}_2) \equiv (d_1 \wedge d_2)(\mathcal{R}_1 \cap \mathcal{R}_2) \equiv (d_1 \wedge d_2)(\mathcal{R}_1) \cap (d_1 \wedge d_2)(\mathcal{R}_2)$
   - $d_1(\mathcal{R}_1) \cup d_2(\mathcal{R}_2) \equiv (d_1 \vee d_2)(d_1(\mathcal{R}_1) \cup d_2(\mathcal{R}_2))$
   - $d(\mathcal{R}_1) - \mathcal{R}_2 \equiv d(\mathcal{R}_1 - \mathcal{R}_2) \equiv d(\mathcal{R}_1) - d(\mathcal{R}_2)$
5. diPs can move from one input of an equijoin to the other input if the columns used in the diP match the columns used in the equi-join condition. For outer-joins, a diP can move only if from the left side of a left outer join (and vice versa). No movement is possible with a full outer join.
   - $d_c(\mathcal{R}_1) \bowtie_{c=e} \mathcal{R}_2 \equiv d_c(\mathcal{R}_1 \bowtie_{c=e} \mathcal{R}_2) \equiv d_c(\mathcal{R}_1) \bowtie_{c=e} d_e(\mathcal{R}_2)$; note here that c and e can be sets of multiple columns, then $c = e$ implies set equality.
6. As we saw in Figure 7 2 where a diP on the `customer_sk` column is being pushed down to the `customer` table, diPs on an inner join can push onto one of its input relations, generalizing the latter half of rule#5. This requires the join input to contain all columns used in the diP, i.e., $d(\mathcal{R}_1 \bowtie \mathcal{R}_2) \equiv d(d(\mathcal{R}_1) \bowtie \mathcal{R}_2)$ iff all columns used by the diP d are available in the relation $\mathcal{R}_1$.

To see these rules in action, note that diPs move in Figure 7 2 using rule#4 twice to pull up past a union and an intersection, rule#5 to move from one join input to another at the top of the expression and rule#6 twice to push to a join input. The example in Figure 2 (left) uses rule#5 at the joins and rule#3 to push below the group-by.

3.4 Scheduling the deriving of predicates

Given a join graph $\mathcal{G}$ where tables are nodes and edges correspond to join conditions, the goal here is to achieve the largest possible data skipping (which improves query performance) while constructing the fewest number of diPs (which reduces QO time).

Consider the example join graphs in Figure 8. The simple case of two tables on the left only requires a single exchange of diPs followed by an update to the partition subsets q; proof is in [19]. The other two cases require more careful handling as we discuss next; the join graph in the middle is the popular star-join which leads to tree-like join graphs and on the right is a cyclic join graph.

Our algorithm to hasten convergence is shown in Pseudocode 9, Scheduler method at line#37. The case of acyclic join graphs is an important sub-case because it applies to queries with star or snowflake schema joins. Here, we construct a tree over the graph (Treeify in line#39 picks the root that has the smallest tree height and sets parent–child pointers; details are in [19]). Then, we pass diPs up the tree (lines#7–#9) and afterwards pass diPs down the tree (lines#10–#12). To see why this converges, note that when line#10 begins, the partition subsets of the table at the root of the tree would have stabilized; Figure 8 (middle) illustrates this case with t_2 as root and shows that convergence requires at most two (parallel) epochs and

Inputs: $\mathcal{G}$, the join graph and $\forall i, q_i$ denoting partitions to be read in table i (notation is listed in Table 3)
Output: $\forall$ tables i, updated q_i reflecting the effect of diPs

```
1  Func: DataPred (q, {c}) // Construct diP for columns {c} over partitions x having q^x =1; see §4.
2  Func: Satisfy (d, x) // = 1 if partition x satisfies predicate d; 0 otherwise. See §4.
3  Func: Exchange(i, j) : //send diP from table i to table j
4  d_{i→j} ← DataPred(q_i, ColsOf(p_{ij}, i))
5  ∀ partition x ∈ table j, q_j^x ← q_j^x * Satisfy(d_{i→j}, x)
6  Func: TreeScheduler(𝒯, {q_i}): // a tree-like join graph
7  for h ← 0 to height(𝒯) − 1 // bottom-up traversal do
8      foreach t ∈ 𝒯 : height(t) = h do
9          Exchange(t, Parent(t, 𝒯))
10 for h ← height(𝒯) to 1 // top-down traversal do
11     foreach t ∈ 𝒯 : height(t) = h do
12         ∀ child c of t in 𝒯, Exchange(t, c)
13 Func: ExchangeExt (𝒢, u, v)//send diPs from node u to v
14 foreach t_1 ∈ RelationsOf(u), t_2 ∈ RelationsOf(v) do
15     if IsConnected(t_1, t_2, 𝒢) then
16         d ← DataPred(q_{t_1}, ColsOf(p_{t_1 t_2}, t_1));
17         ∀ partition x ∈ table t_2, q_{t_2}^x ← q_{t_2}^x * Satisfy(d, x)
18 Func: ProcessNode (u) // exchange diPs within node
19 for i ← 0 to κ // repeat up to κ times do
20     change ← false;
21     foreach tables t_1, t_2 ∈ RelationsOf(u) do
22         if (t_1 ≠ t_2) ∧ IsConnected(t_1, t_2, 𝒢) then
23             d ← DataPred(q_{t_1}, ColsOf(p_{t_1 t_2}, t_1));
24             foreach partition x ∈ t_2 : q_{t_2}^x = 1 do
25                 q_{t_2}^x ← Satisfy(d, x)
26                 change ← change ∨ (q_{t_2}^x = 0)
27     if ¬ change then break// no new pruning;
28 Func: TreeSchedulerExt (𝒱, 𝒢, {q_i})// for cyclic join graphs.
29 for h ← 0 to height(𝒱) − 1 // bottom-up traversal do
30     foreach u ∈ 𝒱 : height(u) = h do
31         if IsNotSingleRelation(u) then ProcessNode(u);
32         ExchangeExt(𝒢, u, Parent(u, 𝒱));
33 for h ← height(𝒱) to 0 // top-down traversal do
34     foreach u ∈ 𝒱 : height(t) = h do
35         if IsNotSingleRelation(u) then ProcessNode(u);
36         ∀ child v of u in 𝒱, ExchangeExt(𝒢, u, v);
37 Func: Scheduler(𝒢, {q_i}):
38 if IsTree(𝒢) then
39     return TreeScheduler (Treeify(𝒢), {q_i});
40 else
41     𝒱 ← MaxWtSpanTree(CliqueGraph(Triangulate(𝒢)))
42     return TreeSchedulerExt(Treeify(𝒱), 𝒢, {q_i});
```

Figure 9: Pseudocode to compute a fast schedule.

six diPs. A proof that this algorithm is optimal, i.e., can skip all skippable partitions in all tables while constructing the fewest number of diPs, is in [19].

We convert a cyclic join graph into a tree over table subsets. The conversion retains completeness; that is, all partitions that can be skipped in the original join graph remain skippable by exchanging diPs only between adjacent table subsets on the tree. Furthermore, on the resulting tree, we apply the same simple schedule that we used above for tree-like join graphs with a few changes. For example, the join graph in Figure 8(right) becomes the following tree: t_1 — $\{t_2, t_3, t_5\}$ — $\{t_3, t_4, t_5\}$ — $\{t_4, t_5, t_6\}$ — t_7. The steps to achieve this conversion are shown in line# 41 and follow from the junction tree algorithm [13, 69, 82]; a step-by-step derivation is

in [19]. On the resulting tree we mimic the strategy used for tree-like join graphs with two key differences. Specifically, at line#42, Treeify picks a root with the lowest height as before. Then, diPs are exchanged from children to parents (lines#29–#32) and from parents to children (lines#33–#36). The key differences between the TreeSchedulerExt and the TreeScheduler methods are: (1) as the ProcessNode method shows, diPs are exchanged until convergence or at most κ times between relations that are contained in a node and (2) we compute multiple diPs when exchanging information between nodes (see ExchangeExt) whereas the Exchange method constructs at most one diP. Figure 8 (right) illustrates the resulting schedule; the root shown in blue is the node containing $\{t_3, t_4, t_5\}$; epochs #2, #3 and #4 invoke ProcessNode on the triangle subsets of tables which have the same color whereas epochs #1 and #5 exchange at most one diP on the edges shown.

Properties of Algorithm 9: For tree-like join graphs, the method shown is optimal (proof in [19]). For a tree-like join graph $\mathcal{G}$ with n tables, this method computes at most $2(n-1)$ diPs (because a tree has $n-1$ edges) and requires $2\text{height}(\mathcal{G})$ (parallel) epochs where tree height can vary from $\lceil \frac{n}{2} \rceil$ to $\lceil \log n \rceil$. For cyclic join graphs the method shown here is approximate; that is, it will not eliminate all partitions that can be skipped. We show by counter-example in [19] that the optimal schedule for a cyclic join graph can require a very large number of diPs; the sub-optimality arises from limiting how often diPs are exchanged between relations within a node (in the ProcessNode method). In §5.4, we empirically demonstrate that our method for cyclic join graphs is a good trade-off between achieving large data skipping and computing many diPs.

4. USING STATISTICS TO BUILD diPs

Data statistics play a key role in constructing data-induced predicates; recall that the three equations 1— 3 use statistics; the specific statistic used determines both the effectiveness and the cost of these operations. An ideal statistic is small, easy to maintain, supports evaluation of a rich class of query predicates and leads to succinct diPs. In this section, we discuss the costs and benefits of well-known data statistics including our new statistic, *range-set*, which our experiments show to be particularly suitable for constructing diPs.

Zone-maps [14] consist of the minimum and maximum value per column per partition and are maintained by several systems today (see Table 1). Each predicate clause listed in §3.2 translates to a logical operation over the zone-maps of the columns involved in the predicate. Conjunctions, disjunctions and negations translate to an intersection, an union or set difference respectively over the partition subsets that match each clause. For string-valued columns, zone-maps are typically built over hash values of the strings and so equality check is also a logical equality, but regular expressions are not supported.

Note that there can be many false positives because a zone map has no information about which values are present (except for the minimum and maximum values).

The diP constructed using zone-maps, as we saw in the example in Table 2b, is a union of the zone-maps of the partitions satisfying the predicate; hence, the diP is a disjunction over non-overlapping ranges. On the table that receives a diP, a partition will satisfy the diP only if there is an overlap between the diP and the zone-map of that partition. Note that there can be false positives in this check as well because no actual data row may have a value within the range that overlaps between the diP and the partition's zone map. It is straightforward to implement these checks efficiently, and our results will show that zone-maps offer sizable I/O savings (Figures 11, 13).

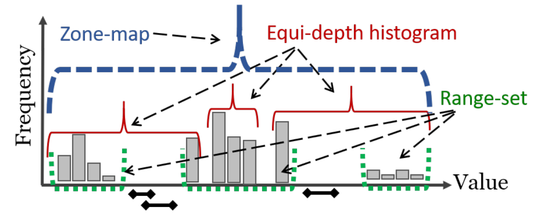


Figure 10: Illustrating the difference between range-sets, zone-maps and equi-depth histogram; both histograms and range-sets have three buckets. The predicates shown in black dumbels below the axes will be false positives for all stats except the range-set.

The false positives noted above do not affect query accuracy but reduce the I/O savings. To reduce false positives, we consider other data statistics.

Equi-depth histograms [48] can avoid some of the false positives when constructed with gaps between buckets. For e.g., a predicate $x = 43$ may satisfy a partition's zone-map because 43 lies between the min and max values for x but can be declared as not satisfied by that partition's histogram if the value 43 falls in a gap between buckets in the histogram. However, histograms are typically built without gaps between buckets [30, 48, 52], are expensive to maintain [52], and the frequency information in histograms, while useful for other purposes, is a waste of space here because predicate satisfaction and diP construction only check for the existence of values.

Bloom filters record set membership [39]. However, we found them to be less useful here because the partition sizes used in practical distributed storage systems (e.g., ~ 100MBs of data [40, 88]) result in millions of distinct values per column in each partition, especially when join columns are keys. To record large sets, bloom filters require large space or they will have a high false positive rate; e.g., a 1KB bloom filter that records a million distinct values will have 99.62% false positives [39] leading to almost no data skipping.

Alternatives such as the count-min [49] and AMS [36] sketches behave similarly to a bloom filter for the purpose at hand. Their space requirement is larger, and they are better at capturing the frequency of values (in addition to set membership). However, as we noted in the case of histograms, frequency information is not helpful to construct diPs.

Range-set: To reduce false-positives while keeping the stat size small, we propose storing a set of non-overlapping ranges over the column value, $\{[l_i, u_i]\}$. Note that a zone-map is a range-set of size 1; using more ranges is hence a simple generalization. The boundaries of the ranges are chosen to reduce false positives by minimizing the total width (i.e., $\sum_i u_i - l_i$) while covering all of the column values. To see why range-sets help, consider the range-set shown in green dots in Figure 10; compared to zone-maps, range-sets have fewer false positives because they record empty spaces or gaps. Equi-depth histograms, as the figure shows, will choose narrow buckets near more frequent values and wider buckets elsewhere which can lead to more false positives. Constructing a range-set over r values takes $O(r \log r)$ time[7]. Reflecting on how zone-maps were used for the three operations in Equations 1– 3, i.e., applying predicates, constructing diPs and applying diPs on joining tables, note that a similar logic extends to the case of a range-set. SIMD-aware implementations can improve efficiency by operating on multiple ranges at once. A range-set having n ranges uses $2n$ doubles. Merging two unsorted range-sets as well as checking for overlap between them

[7] First sort the values, then sort the gaps between consecutive values to find a cutoff such that the number of gaps larger than cutoff is at most the desired number of ranges; see [19] for proof of optimality.

Table 4: Greedily growing a *range-set* in the presence of updates.
Beginning range-set: $\{[3,5],[10,20],[23,27]\}$, $n_r = 3$

	Update	New range-set
order ↓	Add 6	$\{[3,6],[10,20],[23,27]\}$
	Add 13, Delete 20, Change 5 to 15	no change
	Add 52	$\{[3,6],[10,27],[52,52]\}$

uses $O(n \log n)$ time where n is the size of larger rangeset. Our results will show that small numbers of ranges (e.g., 4 or 20) lead to substantial improvements over zone-maps (Figure 16).

4.1 Coping with data updates

When rows are added, deleted, or changed, if the data statistics are not updated, partitions can be incorrectly skipped, i.e., false negatives may appear in equations 1– 3. We describe two methods to avoid false negatives here.

Tainting partitions: A statistic agnostic method to cope with data updates is to maintain a *taint* bit for each partition. A partition is marked as tainted whenever any row in that partition changes. Tables with tainted partitions will not be used to *originate* diPs (because that diP can be incorrect). However, all tables, even those with tainted partitions, can *receive* incoming diPs and use them to eliminate their un-tainted partitions.

Taint bits can be maintained at transactional speeds and can be extremely effective in some cases, e.g., when updates are mostly to tables that do not generate data-reductive diPs. One such scenario is queries over updateable *fact* tables that join with many unchanging *dimension* tables; predicates on dimension tables can generate diPs that flow unimpeded by taint on to the fact tables. Going beyond one taint bit per partition, maintaining taint bits at a finer granularity (e.g., per partition and per column) can improve performance with a small increase in update cost. See results in §5.3.

Approximately updating range-sets in response to updates: The key intuition of this method is to update the range-set in the following approximate manner: ignore deletes and *grow* the range-set to cover the new values; that is, if the new value is already contained in an existing range, there is nothing to do; otherwise, either grow an existing range to contain the new value or merge two consecutive ranges and add the new value as a new range all by itself. Since these options increase the total width of the range-set, the process greedily chooses whichever option has the smallest increase in total width. Table 4 shows examples of greedily growing a *range-set*. Our results will show that such an update is fast (Table 8), and the reduction in I/O savings— because the range-sets after several such updates can have more false positives than range-sets that are re-constructed for just the new column values— is small (Figure 14a).

5. EVALUATION

Using our prototypes in Microsoft's production big-data clusters and SQL server, we consider the following aspects:

- Do data-induced predicates offer sizable gains for a wide variety of queries, data distributions, data layouts and statistic choices?
- Understand the causes for gains and the value of our core contributions.
- Understand the gap from alternatives.

We will show that using diPs leads to sizable gains across queries from TPC-H, TPC-DS and Join Order Benchmark, across different data distributions and physical layouts and across statistics (§5.2). The costs to achieve these gains are small and range-sets offer more gains in more cases than zone-maps (§5.3). Both the careful ordering of diPs and the commutativity rules to move diPs are helpful (§5.4). We also show that diPs are complementary to and sometimes better than using join indexes, materializing denormalized views or clustering rows in §5.5; these alternatives have much higher maintenance costs unlike diPs which work in-situ using small per-table statistics and a small increase to QO time.

5.1 Methodology

Queries: We report results on TPC-H [26], TPC-DS [24] and the join order benchmark (JOB) [60]. We use all 22 queries from TPC-H but because TPC-DS and JOB have many more queries we pick from them 50 and 37 queries respectively[8]. We choose JOB for its cyclic join queries. We choose TPC-DS because it has complex queries (e.g., several non foreign-key joins, UNIONs and nested SQL statements). Query predicates are complex; e.g., q19 from TPC-H has 16 clauses over 8 columns from multiple relations. While inner-joins dominate, the queries also have self-, semi- and outer joins.

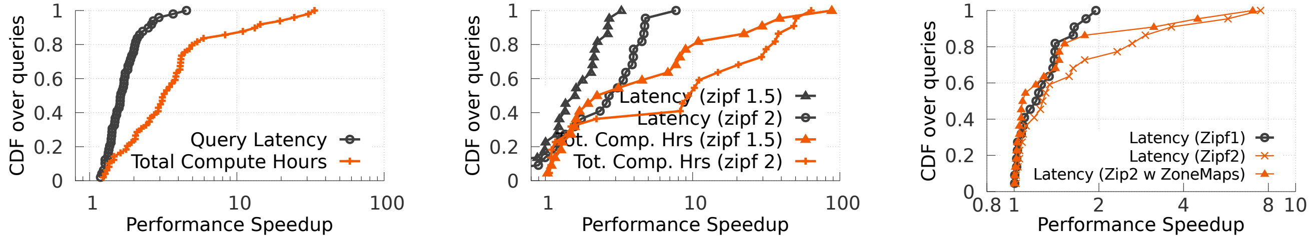
Datasets: For TPC-H and TPC-DS we use 100GB and 1TB datasets respectively. The default datagen for TPC-H, unlike that of TPC-DS, creates uniformly distributed datasets which is not representative of practical datasets; therefore, we also use a modified datagen [29] to create datasets with different amounts of skew (e.g., with zipf factors of 1, 1.5, 2). For JOB, we use the IMDB dataset from May 2013 [60].

Layouts and partitioning: We experiment with many different layouts for each dataset. The tuned layout speeds up queries by avoiding re-partitioning before joins and enhances data skipping[9]. diPs yield sizable gains on tuned layouts. To evaluate behavior more broadly, we generate several other layouts where each table is ordered on a randomly chosen column. For each data layout, we partition the data as recommended by the storage system, i.e., roughly 100MB of content in SCOPE clusters, [45, 88] and roughly 1M rows per columnstore segment in SQL Server [8].

Systems: We have built prototypes on top of two production platforms: SCOPE clusters which serve as the primary platform for batch analytics at Microsoft and comprise tens of thousands of servers [45, 88] and SQL Server 2016. Both systems use cost-based query optimizers [53]. A SCOPE job is a collection of tasks orchestrated by a job manager; tasks read and write to a file system, and each task internally executes a sub-graph of relational operators which pass data through memory. The servers are state-of-the-art Intel Xeons with 192GB RAM, multiple disks and multiple 10Gbps network interface cards. Our SQL server experiments ran on a similar server. After each query executes in SQL server, we flush various system buffer pools to accurately measure the effects of I/O savings. SCOPE clusters use a partitioned row store; for SQL server, we use both columnstores and rowstores. SCOPE and SQL server implement several advanced optimizations such as semijoins [21], predicate pushdown to eliminate partitions [7] and magic-set rewrites [50].

Comparisons: In addition to the above production baselines, we compare against several alternatives. By DenormView, we refer to a technique that avoids joins by *denormalization*, i.e., materializes a join view over multiple tables. The view is stored in column store format in SQL server. Since the view is a single relation, queries can skip partitions without worrying about joins. By JoinIndexes, we refer to a technique that maintains clustered rowstore indexes on the join columns of each relation; for tables that join on more than one column, we build an index on the most frequently used

[8] 1 ... 40, 90 ... 99 from TPC-DS and $([1-9]|10)^*$ from JOB
[9] In short, dimension tables are sorted by key columns and fact tables are clustered by a prevalent predicate column and sorted by columns in the predominant join condition; details are in [19] .



(a) TPC-DS on SCOPE clusters (b) TPC-H (skew=zipf 1.5 or 2) on SCOPE (c) TPC-H (skew=zipf 1 or 2) on SQL server

Figure 11: Change in query performance from using data-induced predicates. The figures show cumulative density functions (CDFs) of speedups for different benchmarks, on different platforms for the tuned data layout (see §5.1). The benefits are wide-spread, i.e., almost all queries improve; in some cases, the improvements can be substantial. More discussion is in §5.2.

join column. By FineBlock, we refer to a single relation workload-aware clustering scheme which enhances data skipping by colocating rows that match or do-not-match the same predicates [78]. We apply FineBlock on the above denormalized view.

We also compare with the following variants of our scheme: No Transforms does not use commutativity to move diPs; Naive Schedule constructs as many diPs as our schedule but picks at random which diP to construct at each step. Preds uses the same statistics but only for predicate pushdown, i.e., it does not compute diPs.

Statistics: Many systems already store zone-maps as noted in Table 1. We evaluate various statistics mentioned in §4. *Gap hist* is our own implementation of an optimal equi-depth histogram with gaps between buckets. Unless otherwise stated, we use 20 ranges for *range-sets* and 10 buckets for *gap hists*. Also, unless otherwise stated the results use *range-sets* to construct diPs.

Metrics: We measure query performance (latency and resource use), statistic size, maintenance costs, and increase in query optimization time. Since diPs reduce the input size that a query reads from the store, we also report InputCut which is the fraction of the query's input that is read after data skipping; if data skipping eliminates half of a query's input, InputCut = 2. When comparing two techniques, we report the ratio of their metric values.

5.2 Benefits from using diPs

Figure 11 shows the performance speedup from using diPs on different workloads in SCOPE clusters and SQL server. Results are on the tuned layout which is popular because it avoids re-partitioning for joins and enhances data skipping [22, 27, 51]. The results are CDFs over queries; we repeat each query at least five times. All of the results except one of the CDFs in Figure 11c use range-sets. Figure 11a shows that the median TPC-DS query finishes almost 2× faster and uses 4× fewer total compute hours. Much larger speed-ups are seen on the tail. Total compute hours improves more than latency (higher speed-up in orange lines than in grey lines) because some of the changes to parallel plans that result from reductions in initial I/O add to the length of the critical path which increases query latency while dramatically reducing total resource use; e.g., replacing pair joins with broadcast joins eliminates shuffles but adds a merge of the smaller input before broadcast [47]. We see that almost all queries improve. SCOPE clusters are shared by over hundreds of concurrent jobs, and so query latency is subject to performance interference; the CDFs use the median value over at least five trials, but some TPC-H queries in Figure 11b still have a small regression in latency. Figures 11b and 11c show that TPC-H queries receive similar latency speedup in SCOPE clusters and SQL server. Unlike TPC-DS and real-world datasets which are skewed, the default datagen in TPC-H distributes data uniformly; these figures show results with different amounts of skew generated using [29].

Table 5: The InputCut from diPs for different benchmarks and different layouts; each query and data layout (see §5.1) contribute a point to a CDF and the table shows values at various percentiles.

Benchmark	InputCut at percentile				
	50th	75th	90th	95th	100th
TPC-H (zipf 2)	1.5×	6.5×	17.7×	32.1×	166.8×
TPC-DS	1.4×	4.1×	7.2×	12.0×	22.4×
JOB	1.9×	2.3×	3.1×	3.4×	115.1×

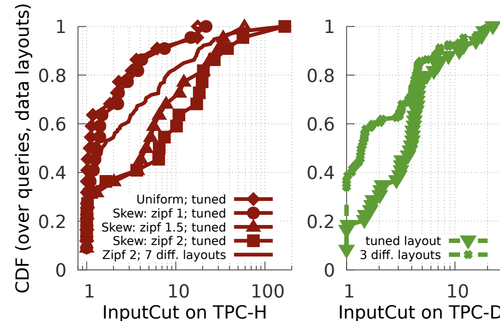


Figure 12: How input skew and data layouts affect the usefulness of diPs; see §5.2.

We see that diPs produce larger speed-ups as skew increases mainly because predicates and diPs become more selective at larger skew. Figure 11c shows sizable latency improvements when using zone-maps. We have confirmed that the query plans in the production systems, SCOPE clusters and SQL server, reflect the effects of predicate pushdown and bitmap filters for semijoins [7, 21, 50]; these figures show that diPs offer sizable gains for a sizable fraction of benchmark queries on top of such optimizations.

Table 5 considers many different layouts, and Figure 12 also considers different skew factors. These results show the InputCut metric which is the reduction in initial I/O read by a query. Across data layouts, about 40% of the queries in each benchmark obtain an InputCut of at least 2×; that is, they can skip over half of the input. Figure 12 shows that lower skew leads to a lower InputCut, but diPs offer gains even for a uniformly distributed dataset. The tuned data layout in both TPC-H and TPC-DS leads to larger values of InputCut relative to the other data layouts; that is, diPs skip more data in the tuned layout. This is because the tuned layouts help with all three conditions C1 – C3 listed in §2; predicates skip more partitions on each table because tuned layouts cluster by predicate columns and ordering by join column values helps diPs eliminate more partitions on the receiving tables. We also observe several instances where a query speeds up more in a different layout than the tuned layout; typically, such queries use different join or predicate columns than those used by the tuned layout.

Figure 13 breaks-down the gains for each query in TPC-H when

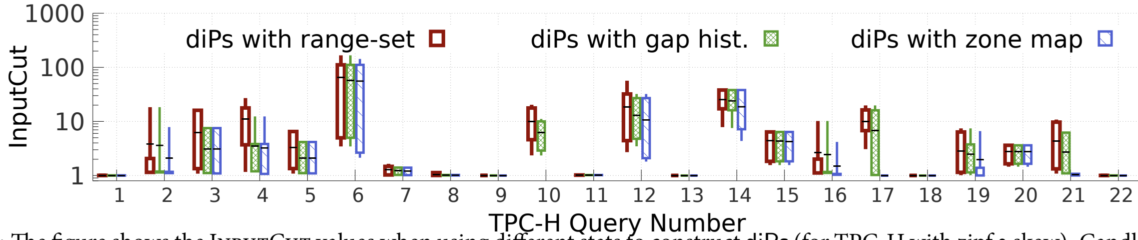


Figure 13: The figure shows the InputCut values when using different stats to construct diPs (for TPC-H with zipf 2 skew). Candlesticks show variation across seven different data layouts including the tuned layout; the rectangle goes from 25th to 75th percentile, the whiskers go from min to max and the thin black dash indicates the average value. Zone-maps do quite well and range-sets are a sizable improvement.

Table 6: The additional latency to derive diPs in seconds compared to the baseline QO latency; see §5.3.

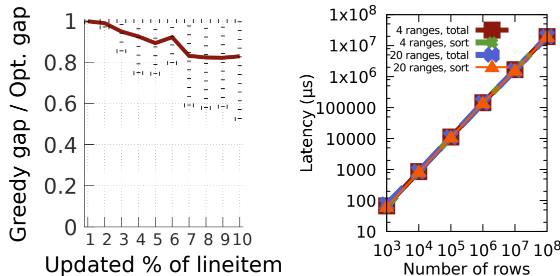
Latency(s) \ %ile	10th	25th	50th	75th	90th
Baseline QO	0.145	0.158	0.176	0.188	0.218
to add diPs	0.032	0.050	0.084	0.107	0.280

Table 7: Additional results for experiments in Figure 11, Table 5 and Figure 12. The table shows data from our SCOPE cluster.

	TPC-H	**TPC-DS**	**JOB**
Input size	100GB	1TB	4GB
#Tables, #Columns	8, 61	24, 416	21, 108
# Queries	22	50	37
Range-set size	~ 2MB	~ 35MB	~ 30KB
# Partitions	~ 10^3	~ $4 * 10^4$	~ 200

Table 8: The time to greedily update range-sets of various sizes (in nanoseconds) measured on a desktop.

Size	2	4	8	16	20	32	64
Avg.	8.5	11.8	22.8	42.1	49.8	67.8	121.4
Stdev.	0.4	0.4	0.4	0.1	2.4	3.4	3.9



(a) Greedy updates. (b) Construction latency

Figure 14: (Left) Effectiveness of greedy-updates for *range-sets*; the figure shows the average and stdev across columns. (Right) Cost to construct *range-sets* measured on a desktop.

using different statistics. Notice that zone-maps are often as good as the gap histograms to construct diPs; compare the third blue candlestick in each cluster with the second green candlestick. Gap histograms are better in predicate satisfaction than zone-maps but do not lead to much better diPs. As the figure shows, range-sets (the first red candlestick in each cluster) offer a marked improvement; they offer larger gains on more queries and in more layouts.

5.3 Costs of using diPs

The costs to obtain this speed-up include storing statistics, an increase to the query optimization duration (to determine which partitions can be skipped), and maintaining statistics when data changes. In big-data clusters, queries are read-only and datasets are bulk appended; so building and maintaining statistics is less impactful relative to the storage space and QO overhead. Table 6 shows that the

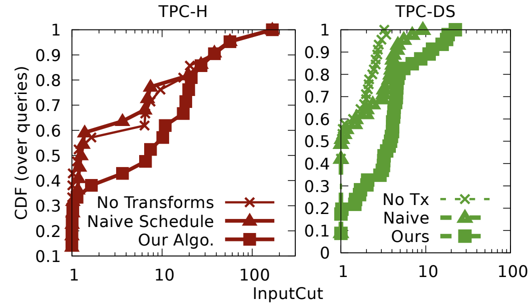


Figure 15: How InputCut varies when different methods are used to derive diPs; we compare with No Transforms which does not use any transformation rules and Naive schedule which constructs the same number of diPs in a naive manner. (Results are for TPC-H skewed with zipf 2 and TPC-DS in the tuned layout.)

additional QO time to use diPs is rather small often, but it can be large on the tail. We verify that these outliers exchange diPs between large tables which takes a long time because such diPs have many clauses and are evaluated on many partitions. We note that our derivation of diPs is a prototype, parts of which are in c# for ease-of-debugging, and that evaluating diPs is embarrassingly parallel (e.g., apply diP to stat of each partition); we have not yet implemented optimizations and believe that the extra QO time can be substantially smaller. Regarding storage overhead, Table 7 shows the size of range-sets which can be thought of as 20 "rows" per partition (a partition is 100MB of data in SCOPE clusters and 1M rows in a columnstore segment in SQL server [8]) and so the space overhead for range-sets is ~ 0.002%. Zone-maps use 10× less space because they only record the max and min value per column, i.e., 2 "rows" per partition. Although TPC-DS and JOB have more tables and more columns, their ratio of stat size to input size is similar.

Costs and gains when tainting partitions: Recall from §4 that a statistic-agnostic method to cope with data updates was to taint partitions. We evaluate this approach by using the TPC-H data generator to generate 100 update sets each of which change 0.1% of the `orders` and `lineitem` tables. Figure 13 showed that diPs deliver sizable gains for 15 out of the 22 TPC-H queries; among these queries, only six are unaffected by taints; specifically for {q2, q14, q15, q16, q17, q19}, diPs offer large I/O savings in spite of updates. The other queries see reduced I/O savings because updates in TPC-H target the two largest tables, `lineitem` and `orders`; when both these relations become tainted, diPs cannot flow *between* these relations, and so queries that require diPs between these tables lose InputCut due to taints. As noted in §4, taints are more suitable when updates target smaller dimension tables.

Greedily maintaining range-sets: Recall from §4 that our second proposal to cope with data updates is to greedily grow the range-set statistic to cover the new values. Table 8 shows that range-sets

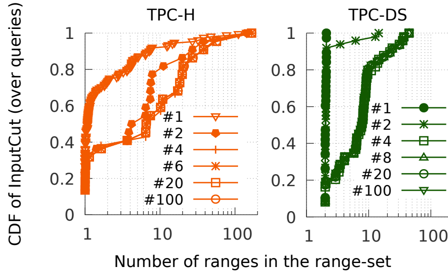


Figure 16: How InputCut varies with the numbers of ranges used in the range-set statistic. (Results are for TPC-H skewed with zipf 2 and TPC-DS in the tuned layout; other cases behave similarly.)

can be updated in tens of nanoseconds using one core on a desktop; thus, the anticipated slowdown to a transaction system is negligible. Figure 14a shows that the greedy update procedure leads to a reasonably high quality range-set statistic; that is, the total gap value (i.e., $\sum_i (u_i - l_i)$ for a range-set $\{[l_i, u_i]\}$) obtained after many greedy updates is close to the total gap value of an optimal range-set constructed over the updated dataset. The figure shows that the greedy updates lead to a range-set with an average gap value ≥ 80% of optimal when up to 10% of rows in the `lineitem` table are updated.

Range set construction time: Figure 14b shows the latency to construct range-sets. Computing larger range-sets (e.g., 20 ranges vs. 4) has only a small impact on latency, and almost all of the latency is due to sorting the input once (the 'total' lines are indistinguishable from the 'sort' lines). These results use std::sort from Microsoft Visual C++. Note that range-sets can be constructed during data ingestion in parallel per partition and column; construction can also piggy-back on the first query to scan the input.

5.4 Understanding why diPs help

Comparing different methods to construct diPs: Figure 15 shows that both the commutativity rules in §3.3 and the algorithm in §3.4 are necessary to obtain large gains using diPs. The naïve schedule has the same QO duration because it constructs the same number of diPs, but by not carefully choosing the order in which diPs are constructed, this schedule leaves gains on the table as shown in the figure. Not using commutativity rules leads to a faster QO time but, as the figure shows, can lead to much smaller performance improvements because generating diPs only for maximal select-join portions of a query graph will not reduce I/O when queries have nested statements and other complex operators. The more complex queries in TPC-DS suffer a greater falloff.

How many ranges to use? Figure 16 shows that a small number of ranges achieve nearly the same amount of data skipping as much larger range-sets. Each step in diP creation, as noted in §4, adds false positives, and there is a limit to gains based on the joint distribution of join and predicate columns. We believe that achieving more I/O skipping beyond that obtained by using just a few ranges may require much larger statistics and/or more complex techniques.

5.5 Comparing with alternatives

Join Indexes: Figure 17 compares using diPs with the JoinIndexes scheme described in §5.1. Results are on SQL server for TPC-H skewed with zipf factor 1 and a scale factor of 100. We built clustered rowstore indexes [6] on the key columns of the dimension tables, and on the fact tables, we built clustered indexes on their most frequently used join columns (i.e., `l_orderkey`, `o_orderkey`, `ps_partkey`). The figure shows that using join indexes leads to worse query latency than not using the indexes in 19/22 queries; we

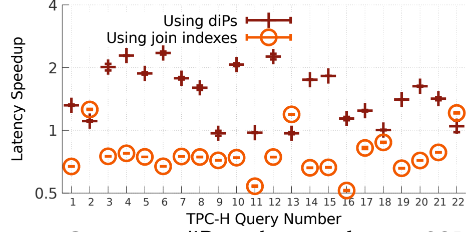


Figure 17: Comparing diPs with join indices on SQL server.

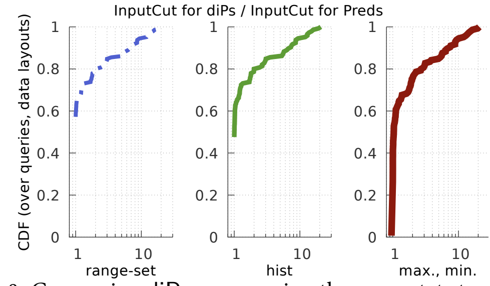


Figure 18: Comparing diPs versus using the same stats to only skip partitions on individual tables.

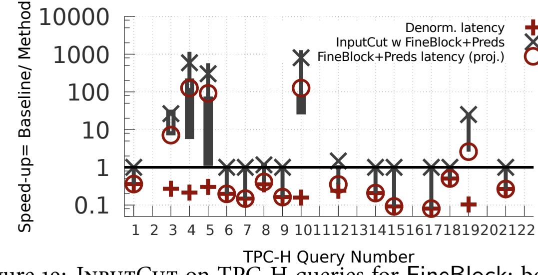


Figure 19: InputCut on TPC-H queries for FineBlock; box plots show results for different predicates.

believe that this is because: (1) the predicate selectivity in several TPC-H queries is not small enough to benefit from an index seek and so most plans use a clustered index scan, and (2) clustered index scans are slower than table scans. diPs are complementary because they reduce I/O before query execution.

diPs vs. predicate pushdown: Figure 18 shows the ratio of improvement over Preds which can only skip partitions on individual tables. When queries have no joins or the selective predicates are only on large relations, diPs do not offer additional data skipping, but the figure shows that diPs offer a marked improvement for a large number of queries and layouts.

diPs vs. DenormView: We use a materialized view (denormalized relation) which subsumes 16/22 queries in TPC-H; the remaining queries require information that is absent in the view and cannot be answered using this view (details are in [19]). In columnar (rowstore) format, this view occupies 2.7× (6.2×) more storage space than all of the tables combined. Queries over the view are unhindered by joins because all predicates directly apply on a single relation; however, because the relation is larger in size, queries may or may not finish faster. We store this view in columnar format and compare against a baseline where all tables are in a columnar layout. Figure 19 (with + symbol) shows the speed-up in query latency when using this view in SQL server (results are for 100G dataset with zipf skew 1). We see that all of the queries slow down (all + symbols are below 1). Materialized views speed-up queries, in general, only when the views have selections or aggregations that reduce the size of the view [34]. Unfortunately, this does not happen in the case of the view that subsumes all 16/22 TPC-H queries (see [19]).

diPs vs. clustering rows by predicates: A recent research proposal [78] clusters rows in the above view to maximize data skipping. Training over a set of predicates, [78] learns a clustering scheme that intends to skip data for unseen predicates. Figure 19 shows with × symbols the average InputCut obtained as a result of such clustering; the candlesticks around the × symbol show the min, 25th percentile, 75th percentile and max InputCut for different query predicates. We see that most queries receive no InputCut (× marks are at 1) due primarily to two reasons: (1) the chosen clustering scheme does not generalize across queries; that is, while some queries receive gains the chosen clustering of rows does not help all queries, and (2) the chosen clustering scheme does not generalize to unseen predicates as can be seen from the large span of the candlesticks. Figure 19 also shows with circle symbols the average query latency when using this clustering. Only 5/22 queries improve (fastest query is ~ 100× faster) and 11/22 queries regress (slowest is 10× slower). Hence, the practical value of such schemes is unclear.

We also note the rather large overheads to create and maintain indexes and views [33, 70] and to learn clusterings [78]. Also, these schemes require foreknowledge of queries and offer gains only for queries that are similar [34, 35, 78]. In contrast, diPs only use small and easily maintainable data statistics, require no apriori knowledge of queries and offer sizable gains for ad-hoc and complex queries.

6. RELATED WORK

To the best of our knowledge, this paper is the first system to skip data across joins for complex queries during query optimization. These are fundamental differences: diPs rely only on simple per-column statistics, are built on-the-fly in the QO, can skip partitions of multiple joining relations, support different join types and work with complex queries; the resulting plans only read subsets of the input relations and have no execution-time overhead.

Some research works discover data properties such as functional dependencies and column correlations and use them to improve query plans [10, 42, 56, 58]. Inferring such data properties is a sizable cost (e.g., [56] uses student t-test between every pair of columns). It is unclear if these properties can be maintained when data evolves. More importantly, imprecise data properties are less useful for QO (e.g., a *soft* functional dependency does not preserve set multiplicity and hence cannot guarantee correctness of certain plan transformations over group-bys and joins). A SQL server option [10] uses the fact that the `l_shipdate` attribute of `lineitem` is between 0 to 90 days larger than `o_orderdate` from `orders` [26] to convert predicates on `l_shipdate` to predicates on `o_orderdate` and vice versa. Others discover similar constraints more broadly [42, 58]. In contrast, diPs exploit relationships that may only hold conditionally given a query and a data-layout. Specifically, even if the predicate columns and join columns are independent, diPs can offer gains if the subset of partitions that satisfy a predicate contain a small subset of values of the join columns. As we saw in §2, such situations arise when datasets are clustered on time or partitioned on join columns [51].

Prior work moves predicates around using column equivalence and magic-set style reasoning [50, 61, 63, 72, 81, 83]. SCOPE clusters and SQL server implement such optimizations, and as we saw in §5, diPs offer gains over these baselines. Column equivalence does not help when predicate columns do not exist in joining relations. Magic set transformations help only 2/22 queries in TPC-H queries and only when predicates are selective [72]. By inferring new predicates that are induced by data statistics, diPs have a wider appeal.

Auxiliary data structures such as views [28], join indices [23], join bitmap indexes [5], succinct tries [87], column sketches [54], and partial histograms [84] can also help speed-up queries. Join zone maps [15] on a fact table can be constructed to include predicate columns from dimension tables; doing so effectively creates zone-maps on a larger denormalized view. Constructing and maintaining these data structures has overhead, and as we saw in §5, a particular view or join index does not subsume all queries. Hence, many different structures are needed to cover a large subset of queries which further increases overhead. Queries with foreign-key — foreign-key joins (e.g., `store_sales` and `store_returns` in TPC-DS join in six different ways) can require maintaining many different structures. diPs can be thought off as a complementary approach that helps with or without auxiliary structures.

While data-induced predicates are similar to the implied integrity constraints used by [65], there are some key differences and additional contributions. (1) [65] only exchanges constraints between a pair of relations; we offer a method which exchanges diPs between multiple relations, handles cyclic joins and supports queries having group-by's, union's and other operations. (2) [65] uses zone maps and two bucket histograms; we offer a new statistic (range-set) that performs better. (3) [65] shows no query performance improvements; we show speed-ups in both a big-data cluster and a DBMS. (4) [65] offers no results in the presence of data updates; we design and evaluate two maintenance techniques that can be built into transactional systems.

While a query executes, sideways information passing (SIP) from one sub-expression to a joining sub-expression can prune the data-in-flight and speed up joins [38, 57, 63, 68, 72, 75]. Several systems, including SQL server, implement SIP and we saw in §5 that diPs offer additional speed-up. This is because SIP only applies during query execution whereas diPs reduce the I/O to be read from the store. SIP can reduce the cost of a join, but constructing the necessary info at runtime (e.g., a bloom filter over the join column values from one input) adds runtime overhead, needs large structures to avoid false positives and introduces a barrier that prevents simultaneous parallel computation of the joining relations. Also, unlike diPs, SIP cannot exchange information in both directions between joining relations nor does it create new predicates that can be pushed below group-bys, unions and other operations.

A large area of related work improves data skipping using workload aware adaptations to data partitioning or indexing [43, 44, 55, 62, 66, 71, 74, 78, 79, 86]; they co-locate data that is accessed together or build correlated indices. Some use denormalization to avoid joins [78, 86]. In contrast, diPs require no changes to the data layout and no foreknowledge of queries.

7. CONCLUSION

As dataset sizes grow, human-digestible insights increasingly use queries with selective predicates. In this paper, we present a new technique that extends the gains from data skipping; the predicate on a table is converted into new data-induced predicates that can apply on joining tables. Data-induced predicates (diPs) are possible, at a fundamental level, because of implicit or explicit clustering that already exists in datasets. Our method to construct diPs leverages data statistics and works with a variety of simple statistics, some of which are already maintained in today's clusters. We extend the query optimizer to output plans that skip data before query execution begins (e.g., partition elimination). In contrast to prior work that offers data skipping only in the presence of complex auxiliary structures, workload-aware adaptations and changes to query execution, using diPs is radically simple. Our results in a large data-parallel cluster and a DBMS show that large gains are possible across a wide variety of queries, data distributions and layouts.

8. REFERENCES

[1] 2017 big-data and analytics forecast. https://bit.ly/2TtKyjB.
[2] Apache orc spec. v1. https://bit.ly/2J5BIkh.
[3] Apache spark join guidelines and performance tuning. https://bit.ly/2Jd87We.
[4] Band join. https://bit.ly/2kixJJn.
[5] Bitmap join indexes in oracle. https://bit.ly/2TLBBTF.
[6] Clustered and nonclustered indexes described. https://bit.ly/2Drdb9o.
[7] Columnstore index performance: Rowgroup elimination. https://bit.ly/2VFpljV.
[8] Columnstore indexes described. https://bit.ly/2F7LZuI.
[9] Data skipping index in spark. https://bit.ly/2qONacb.
[10] Date correlation optimzation in sql server 2005 & 2008. https://bit.ly/2VodSVN.
[11] Imdb datasets. https://imdb.to/2S3BzSF.
[12] Join order benchmark. https://bit.ly/2tTRyIb.
[13] The junction tree algorithm. https://bit.ly/2lPHNtA.
[14] Oracle database guide: Using zone maps. https://bit.ly/2qMeO9E.
[15] Oracle: Using zone maps. https://bit.ly/2vsUWKK.
[16] Parquet thrift format. https://bit.ly/2vm6D5U.
[17] Presto: Repartitioned and replicated joins. https://bit.ly/2JauYll.
[18] Processing petabytes of data in seconds with databricks delta. https://bit.ly/2Pryf2E.
[19] Pushing data-induced predicates through joins in bigdata clusters; extended version. https://bit.ly/2WhTwP1.
[20] Query 17 in tpc-h, see page #57. https://bit.ly/2kJRV72.
[21] Query execution bitmap filters. https://bit.ly/2NJzzgF.
[22] Redshift: Choosing the best sort key. https://amzn.to/2AmYbXh.
[23] Teradata: Join index. https://bit.ly/2FbalDT.
[24] TPC-DS Benchmark. http://www.tpc.org/tpcds/.
[25] Tpc-ds query #35. https://bit.ly/2U0rIk6.
[26] TPC-H Benchmark. http://www.tpc.org/tpch.
[27] Vertica: Choosing sort order: Best practices. https://bit.ly/2yrvPtG.
[28] Views in sql server. https://bit.ly/2CnbmIo.
[29] Program for tpc-h data generation with skew. https://bit.ly/2wvdNVo, 2016.
[30] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *SIGMOD*, 1999.
[31] P. K. Agarwal et al. Mergeable summaries. *TODS*, 2013.
[32] S. Agarwal et al. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.
[33] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient view maintenance at data warehouses. In *SIGMOD*, 1997.
[34] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. *VLDB*, 2000.
[35] S. Agrawal et al. Database tuning advisor for microsoft sql server 2005. *VLDB*, 2004.
[36] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *STOC*, 1999.
[37] M. Armbrust et al. Spark sql: Relational data processing in spark. In *SIGMOD*, 2015.
[38] F. Bancilhon et al. Magic sets and other strange ways to implement logic programs. In *SIGMOD*, 1985.
[39] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 1970.
[40] D. Borthakur et al. Hdfs architecture guide. *Hadoop Apache Project*, 2008.
[41] D. Borthakur et al. Apache hadoop goes realtime at facebook. In *SIGMOD*, 2011.
[42] P. G. Brown and P. J. Haas. Bhunt: Automatic discovery of fuzzy algebraic constraints in relational data. In *VLDB*, 2003.
[43] M. Brucato, A. Abouzied, and A. Meliou. A scalable execution engine for package queries. *SIGMOD Rec.*, 2017.
[44] L. Cao and E. A. Rundensteiner. High performance stream query processing with correlation-aware partitioning. *VLDB*, 2013.
[45] R. Chaiken et al. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *VLDB*, 2008.
[46] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 2012.
[47] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD*, 2015.
[48] G. Cormode, M. Garofalakis, P. J. Haas, C. Jermaine, et al. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 2011.
[49] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 2005.
[50] M. Elhemali, C. A. Galindo-Legaria, T. Grabs, and M. M. Joshi. Execution strategies for sql subqueries. In *SIGMOD*, 2007.
[51] M. Y. Eltabakh et al. Cohadoop: Flexible data placement and its exploitation in hadoop. In *VLDB*, 2011.
[52] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. In *VLDB*, 1997.
[53] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 1995.
[54] B. Hentschel, M. S. Kester, and S. Idreos. Column sketches: A scan accelerator for rapid and robust predicate evaluation. In *SIGMOD*, 2018.
[55] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, 2007.
[56] I. Ilyas et al. Cords: Automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, 2004.
[57] Z. G. Ives and N. E. Taylor. Sideways information passing for push-style query processing. In *ICDE*, 2008.
[58] H. Kimura et al. Correlation maps: a compressed access method for exploiting soft functional dependencies. In *VLDB*, 2009.
[59] A. Lamb et al. The vertica analytic database: C-store 7 years later. *VLDB*, 2012.
[60] V. Leis et al. How good are query optimizers, really? In *VLDB*, 2015.
[61] A. Y. Levy, I. S. Mumick, and Y. Sagiv. Query optimization by predicate move-around. In *VLDB*, 1994.
[62] Y. Lu, A. Shanbhag, A. Jindal, and S. Madden. AdaptDB: Adaptive partitioning for distributed joins. In *VLDB*, 2017.
[63] I. S. Mumick and H. Pirahesh. Implementation of magic-sets in a relational database system. In *SIGMOD*, 1994.
[64] A. Nanda. Oracle exadata: Smart scans meet storage indexes. http://bit.ly/2ha7C5u, 2011.

[65] A. Nica et al. Statisticum: Data Statistics Management in SAP HANA. In *VLDB*, 2017.
[66] M. Olma et al. Slalom: Coasting through raw data via adaptive partitioning and indexing. In *VLDB*, 2017.
[67] C. Olston et al. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, 2008.
[68] J. M. Patel et al. Quickstep: A data platform based on the scaling-up approach. In *VLDB*, 2018.
[69] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.
[70] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *SIGMOD*, 1996.
[71] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The uncracked pieces in database cracking. In *VLDB*, 2013.
[72] P. Seshadri et al. Cost-based optimization for magic: Algebra and implementation. In *SIGMOD*, 1996.
[73] A. Shanbhag et al. A robust partitioning scheme for ad-hoc query workloads. In *SOCC*, 2017.
[74] A. Shanbhag, A. Jindal, Y. Lu, and S. Madden. Amoeba: a shape changing storage system for big data. In *VLDB*, 2016.
[75] L. Shrinivas et al. Materialization strategies in the vertica analytic database: Lessons learned. In *ICDE*, 2013.
[76] J. Shute et al. F1: A distributed sql database that scales. In *VLDB*, 2013.
[77] D. Ślęzak et al. Brighthouse: An analytic data warehouse for ad-hoc queries. In *VLDB*, 2008.
[78] L. Sun et al. Fine-grained partitioning for aggressive data skipping. In *SIGMOD*, 2014.
[79] L. Sun, M. J. Franklin, J. Wang, and E. Wu. Skipping-oriented partitioning for columnar layouts. In *VLDB*, 2017.
[80] A. Thusoo et al. Hive- a warehousing solution over a map-reduce framework. In *VLDB*, 2009.
[81] N. Tran et al. The vertica query optimizer: The case for specialized query optimizers. In *ICDE*, 2014.
[82] M. J. Wainwright and M. I. Jordan. Graphical models, exponential families, and variational inference. `https://bit.ly/2yurPIS`, 2008.
[83] B. Walenz, S. Roy, and J. Yang. Optimizing iceberg queries with complex joins. In *SIGMOD*, 2017.
[84] J. Yu and M. Sarwat. Two birds, one stone: a fast, yet lightweight, indexing scheme for modern database systems. In *VLDB*, 2016.
[85] M. Zaharia et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.
[86] E. Zamanian, C. Binnig, and A. Salama. Locality-aware partitioning in parallel database systems. In *SIGMOD*, 2015.
[87] H. Zhang et al. Surf: Practical range query filtering with fast succinct tries. In *SIGMOD*, 2018.
[88] J. Zhou et al. SCOPE: Parallel databases meet MapReduce. In *VLDB*, 2012.
[89] J. Zhou, P.-A. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the scope optimizer. In *ICDE*, 2010.