Pushing Data-Induced Predicates Through Joins in Big-Data Clusters : Extended Version

Microsoft

Abstract– Using data statistics, we convert predicates on a table into data-induced predicates (diPs) that can 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.

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.

```
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 [50], magic set rewriting [47, 72] or value-based pruning [81]), but predicates over join columns are exceedingly rare,[1] and these techniques do not apply when the predicates use columns that do not exist in the joining tables.

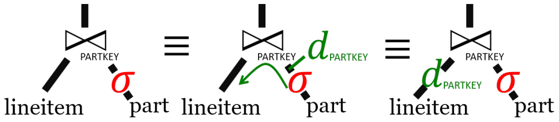


Figure 1: Example illustrating creation and use of a data-induced predicate which only uses the join columns and is a necessary condition to the true predicate, i.e., $\sigma \Rightarrow d_{\texttt{partkey}}$.

Some systems implement a form of sideways information passing over joins [18, 66] 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 causing the scan of lineitem to wait 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 minimalistic assumptions.

Our target scenario is big-data systems, e.g., SCOPE [42], Spark [34, 85], Hive [79] or Pig [65] 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 basic idea is as follows; an example is shown in Figure 1. First, we use data statistics to eliminate partitions on tables that have predicates. This step is standard and is already implemented in some systems [6, 16, 42, 81]. Next,

Proceedings of the VLDB Endowment, Vol. 13, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

[1] Over all the queries in TPC-H [80] and TPC-DS [25], there are zero predicates on join columns perhaps because join columns tend to be opaque system-generated identifiers.

Scheme	Statistic	Granularity
ZoneMaps [12]	max and min value per column	*zone*
Spark [8, 16]		*file*
Exadata [62]	max, min and null present or null count per column	per table *region*
Vertica [56], ORC [2], Parquet [14]		stripe, rowgroup
Brighthouse [76]	histograms, char maps per col	*data pack*

Table 1: Data statistics maintained by several systems.

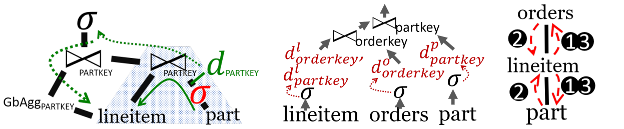


Figure 2: Illustrating the need to move diPs past other operations (left). On a 3-way join when all tables have predicates (middle), the optimal schedule only requires three (parallel) steps (right).

on the partitions that satisfy the local predicates, we use their data statistics to construct a new predicate which captures all join column values contained in these 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., not all rows in the partitions included in the diP may satisfy σ. However, the diP can apply over the joining table because it only uses the join column[2]. The diP can be applied on the data statistics of the joining table, `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 [44]) 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 [43] or using join indexes [4, 21]. The fundamental reason is that these techniques use augmentary data-structures which are costly to maintain and yet their benefits are limited to narrow classes of queries (e.g., queries that match views, have specific join conditions, or very selective predicates) [31]. 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 [48, 60] or cluster rows that satisfy the same predicates [73, 77]; 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 [18, 66, 50, 81, 47, 72] or uses more complex structures which have sizable maintenance overheads [43, 4, 21, 77, 73]. 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, *range-set*, that improves query performance over zone-maps at the cost of a small increase in space. We also discuss techniques to maintain statistics when datasets evolve. In more detail, the rest of this paper has these contributions.

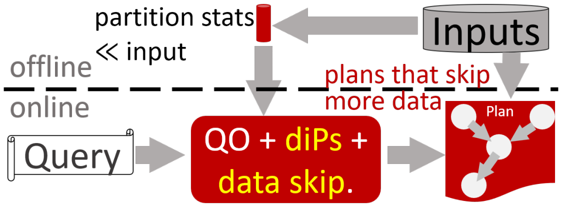


Figure 3: A workflow which shows changes in red; using partition statistics, our query optimizer computes data-induced predicates and outputs plans that read less input.

- Using diPs for complex query expressions leads to novel challenges. Consider TPC-H q17 which as shown in Figure 2(left) has two operations over the `lineitem` table. Creating diPs for only the fragment considered in Figure 1 still reads the entire `lineitem` table for the other operation. To alleviate this, we use new QO transformation rules to move diPs; in this case, shown with dotted arrows, the diP moves past two joins and a group-by and ensures that a single 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 any partition on a joining table, then the previously constructed diPs from that table are no longer up-to-date. Re-creating diPs whenever partition subsets change will increase data skipping gains; however, doing so naïvely can construct excessively many diPs which increases query optimization time. By establishing an analogy with inference on graphs, we present an optimal method for tree-like join graphs which converges to fixed point and hence achieves all possible data skipping while computing the fewest number of diPs. Joins in star and snowflake schemas are tree-like. Our solution for general join graphs and how we derive diPs within a cost-based query optimizer is 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, maintainability and potential for data skipping.
- We discuss two methods to cope with dataset updates in §4.1. The first method *taints* a partition when any row in that partition changes; tainted partitions are never skipped; tables that contain tainted partitions cannot originate diPs, but they can use diPs received from joining tables to eliminate untainted partitions. Our second method approximately updates data statistics by ignoring deletes and *growing* the statistic to cover new values. We will show in §5 that typical *range-sets* can be updated in tens of nanoseconds and that their usefulness decays gracefully as larger portions of the tables are updated.

[2] We call this a data-induced predicate because it is specific to the query predicate as well as the data statistics of the table that the predicate applies upon.

- 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 several real-world use-cases that lead to this property holding in practice 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 [80], TPC-DS [25], JOB [11]) 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, $\sigma_{\texttt{d\_year}}$ (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 d_year is only available on the date_dim table, but the diPs are on the join column d_date_sk and can be pushed onto joining relations using column equivalence [50]. 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, (3) use the merged statistic to extract a new predicate and identify partitions that satisfy this predicate in joining relations. Many data statistics support these steps [28], and different stats can be used for different steps.

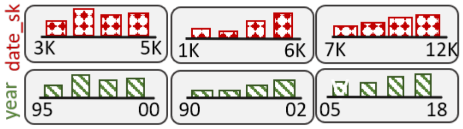
Column	Partition #		
	1	**2**	**3**
d_date_sk	$[3000, 5000]$	$[1000, 6000]$	$[7000, 12000]$
d_year	$[1995, 2000]$	$[1990, 2002]$	$[2005, 2018]$

(a) Zone maps [12], i.e., 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 $\leq$ 1995	$\{1, 2\}$	date_sk $\in$ $[1000, 6000]$	45%
year $\in$ $[2003, 2004]$	$\varnothing$	date_sk $\in$ $[]$	0%
year $>$ 2010	$\{3\}$	date_sk $\in$ $[7000, 12000]$	45%

(b) Data-induced predicates on the join column d_date_sk corresponding to predicates on the column d_year.

Table 2: Constructing diPs using partition statistics.



(a) Equiwidth histograms for the above example.

Range-set (size 2)	Partition #		
	1	**2**	**3**
d_date_sk	$\{[3000, 3500],$ $[4000, 5000]\}$	$\{[1000, 2000],$ $[5000, 6000]\}$	$\{[7000, 10000],$ $[11000, 12000]\}$
d_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: Showing other data statistics (histograms, range-sets) for the same example as in Table 2a.

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[3] 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 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[4]. 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 the TPC-H query #17 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 [26], and tables were laid out in a typical manner[5]. Each partition is $\sim$ 100MBs of data which is a typical quanta in distributed file systems [37] and is the default in our clusters [88]. Even though 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 the vast majority of the 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.

2.1 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.

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

[4] For year $\leq$ 1995, the diP using two ranges is date_sk $\in$ $\{[1000, 2000], [3000, 3500], [4000, 6000]\}$ which covers 30% fewer values than the diP constructed using a zone-map, date_sk $\in$ $[1000, 6000]$.

[5] part was sorted on its key; lineitem was clustered on l_shipdate and each cluster sorted on l_orderkey; this layout is known to lead to good performance because it reduces re-partitioning for joins while allowing date-based predicates to skip partitions [3, 19, 23].

#Partitions remaining in...

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

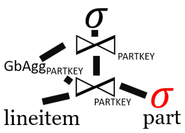


Figure 5: For TPC-H query 17 in Figure 2 (left), the table shows the partition reduction from using diPs. On the right, we show an equivalent plan generated using magic-set transformations; group-by is pushed above the join. Note that magic-sets and diPs have complementary value: this magic-set transformation does not allow data skipping on `lineitem` but it simplifies the usage of diPs since the movements shown in Figure 2 (left) are not needed.

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.

C3: In tables that receive diPs, the join column values are distributed such that the diP only picks a small subset of partitions of the receiving table.

identify use-cases where these conditions hold based on our experiences in production clusters at Microsoft [42].

- A majority of the datasets in production clusters are stored in the same order that the data was ingested into the cluster [34, 38]. 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.

- 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, 19, 23, 48]. Performance improves by reducing the shuffles needed to re-partition for joins [15, 89, 44] and by ensuring data skipping for predicates. Such data layouts help with all three conditions C1–C3 and, in our experiments, translate to 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 [10] 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 [29]. When skew is large, both predicates and diPs can become more selective by skipping over heavy-hitters; hence, skew can help C1–C3.

The net effect of the above cases is that the three conditions hold often allowing diPs to enhance data skipping on joining relations.

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 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); 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).

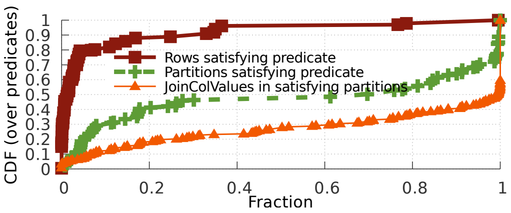


Figure 6: Quantifying how often the conditions that lead to large I/O skipping gains from using diPs hold in practice by using queries and datasets from production clusters at Microsoft.

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.
$d_{i\to j}$	Derived predicate from table i to table j (note: derived predicates are not symmetric)
partition	granularity at which the storage layer maintains data statistics (see Table 1)

Table 3: Notation used in this paper.

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 only uses data statistics over the tables.

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 1, these steps can be written as:

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

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

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

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 (Eq. 1) executes once, but the latter two steps are recursive and 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 Eq. 3), then the diPs from that table (which are computed in Eq. 2 based on q) will have to be re-computed. 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. 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 other relational operators can interfere with the simple method described above. That is, if the query expression consists only of select and join operations, the above method suffices. But, typical query expressions contain many other operations such as group-bys and nested statements. One option is to ignore other operations and apply diPs only to subportions 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 restricting diPs to just the select-join portion (shown with a shaded background in the figure) may lead to no gains if the whole of lineitem table is read for the group-by. To address this challenge, we move diPs around other relational operators using commutativity. We list transformation rules in §3.3 that cover a broad class of operators. Using these transformation rules extends the usefulness of diPs to more 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 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 a query rewriter, separate from the QO, insert optimal diPs into the query? Then, the QO only needs minimal changes. This option is problematic because such a query rewriter has to implement complex logic that is already available within the QO. For example, the rewriter will have to implement predicate simplifications to identify parts of a query predicate that can apply on each table (the p_i's in Eq. 1); as well, the rewriter will have to implement logic to move diPs around other operators (e.g., the rules in §3.3). (2) Can derivation of diPs be implemented as new plan transformation rules? If possible, this would be a small software change because the QO framework can remain unchanged. Unfortunately, diPs are sometimes 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, such recursion and fine-grained query-wide ordering are challenging to achieve [50]. 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 tables, etc. diPs are exchanged first within these maximal select-join sub-portions of the query expression using the schedule in §3.4. Next, diPs are exchanged with the rest of the query using the rules in §3.3. 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. We also support one-sided outer joins (rule#5 in §3.3) and semi, anti-semi joins which are akin to set intersection and set difference respectively (rule#4 in §3.3).

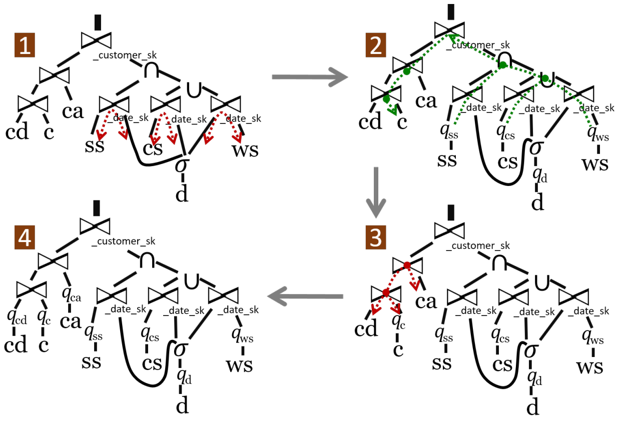


Figure 7: Illustrating deriving predicates for TPC-DS query 35. The table labels ss, cs, ws and d correspond to the tables store_sales, catalog_sales, web_sales and date_dim.

We use transformations for diPs beyond those that are traditionally used for filter operators: new rules inject diPs into the plan, and, after the transformations in §3.3 have been applied, new rules convert diPs that apply directly on tables into reads of partition subsets (i.e., q_i) of the tables. diPs that do not directly apply on a table are removed from the plan. We also use rules that add redundant diPs such as the union rule (#4 in §3.3); such rules are not used for filters because the alternative is costlier, but as we show in the example next, these rules help diPs to move and lead to better plans. Finally, note that this phase executes exactly once for a query. The increase in query optimization time is small, and by applying exploration rules later, the QO can identify plans that benefit from the reduced input sizes (e.g., different join orders or using broadcast joins instead of hashjoins).

We illustrate this overall process using an example.

Example: Figure 7 shows this process in action for TPC-DS query 35; the full SQL statement is in [22]. Data-induced predicates are computed first for each maximal SJ expression store_sales $\bowtie$ date_dim, catalog_sales $\bowtie$ date_dim and web_sales $\bowtie$ date_dim, as shown in the top left of the figure labeled 1. As the plan shows, the result of these expressions joins with another expression on the customer_sk column 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 on an SJ expression ensues.[6] The final plan, shown in 4, effectively replaces each table with the corresponding partition subset vector that has to be read from that table.

3.2 Supported Queries

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

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 using columns from a table:

- 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 and op is either $=, <, \leq, >, \geq, \neq$.
- For string and categorical columns, equality check only.

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) is straightforward. We support inner, one-sided outer, semi, anti-semi and self joins.

Projections: On columns that are used in the above join conditions and predicates, only invertible projections (e.g., $\pi(x) = ax + b$ for column x where a, b are constants) commute with diPs on that column because only such projects can be inverted though zone-maps and other data statistics that we use to compute diPs. Arbitrary projections are supported on other columns.

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. In some cases, the alternatives are costlier (because they use redundant predicates), but we use them to facilitate movement of diPs. As noted in §3.1, 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.
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 derived predicate can skip only if from the left side of a left outer join (and vice versa). No skipping is possible for 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 $\mathcal{R}_1$.

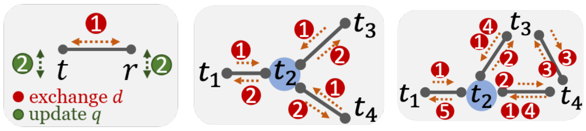


Figure 8: The optimal schedules of exchanging diPs for different join graphs; numbers-in-circles denote the epoch; multiple diPs are exchanged in parallel in each epoch. Details are in §3.4.

To see these rules in action, note that the diP movement in Figure 7 2 uses 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 uses rule#6 twice to push to a join input. The example in Figure 2 (left) uses rule#5 at the above join 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).

To build intuition, consider the examples in Figure 8. The simple case of two tables only requires a single exchange of diPs followed by an update to the partition subsets q; proof is in §10 . Next, the join graph in the middle with four tables in a star join requires six diPs in two (parallel) epochs. Any schedule where diPs *leave* t_2 before diPs from every neighbor go *into* t_2 will take more epochs or construct more diPs because t_2's partition subset stabilizes only after receiving diPs from all of its neighbors. Finally, the join cycle graph on the right also has four tables but is slower to converge requiring ten diPs and five epochs. The key insight here is that some partition in each table may be eliminated only by the joint effect of all of the neighboring diPs and the local predicate but cannot be eliminated by any sub-combination of these predicates. Here, partition subsets stabilize first for tables $\{t_3, t_4\}$ at the end of third epoch because by then they receive all pertinent information. Notice also that the schedule avoids constructing diPs that are not needed for convergence (e.g., t_4 is silent in epochs 2, 5) and constructs diPs on some edges multiple times (twice on $t_3 \rightarrow t_2$).

Our algorithm to hasten convergence is shown in Pseudocode 9, line#17, Scheduler method. For ease of exposition, consider the case of a join graph that has no cycles. This is an important subcase because it applies to queries with star or snowflake schema joins. Here, we construct an implicit tree over the graph (Treeify in line#19), pass diPs up the tree (lines#7–#9) and then down the tree (lines#10–#12). To see why this converges, note that before line#10 executes, 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. For a join graph with n tables, note that this method computes at most $2(n-1)$ diPs (because a tree has $n-1$ edges) and requires $\theta(\mathsf{depth}(\mathcal{G}))$ epochs where tree depth can vary from $\lceil \frac{n}{2} \rceil$ to $\lceil \log n \rceil$.

For join graphs that are not trees, we only briefly sketch our method here. When the size of the largest clique in the join graph is small, we use a modified version of the junction tree algorithm [67]; see line#19 in Figure 9. Intuitively, this algorithm replaces each clique with a new virtual node and adds edges to retain connectivity as before. The process recurses until no cliques remain. Then, the above tree scheduler can be used with the caveat that receiving and constructing diPs at a virtual node require exchanging diPs

```
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
data-induced predicates

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  $d_{i \to j}$ = DataPred$(q_i$, ColsOf$(p_{ij}, t))$
5  $\forall$ partition $x \in$ table $j, q_j^x = q_j^x *$ Satisfy$(d_{i \to j}, x)$

6  Func: TreeScheduler$(\mathcal{T}, \{q_i\})$: // a tree-like join graph
7  for $h \leftarrow 0$ to height$(\mathcal{T}) - 1$ // bottom-up traversal do
8      foreach $t \in \mathcal{T}$ : height$(t) == h$ do
9          Exchange$(t$, parent$(t, \mathcal{T}))$

10 for $h \leftarrow$ height$(\mathcal{T})$ to 1 // top-down traversal do
11     foreach $t \in \mathcal{T}$ : height$(t) == h$ do
12         $\forall$ child $c$ of $t$ in $\mathcal{T}$, Exchange$(t, c)$

13 Func: JunctionTree $(\mathcal{G}, \{q_i\})$
14 // construct a tree-like graph consisting of virtual nodes that
   represent sub-graphs; see [82]

15 Func: CycleScheduler $(\mathcal{G}, \{q_i\})$
16 // mimic approximate inference; see [82].

17 Func: Scheduler$(\mathcal{G}, \{q_i\})$:
18 if IsTree$(\mathcal{G})$ then return TreeScheduler (Treeify$(\mathcal{G}), \{q_i\})$;
19 if LargestClique$(\mathcal{G}) < \kappa$ then return JunctionTree
   $(\mathcal{G}, \{q_i\})$;
20 return CycleScheduler$(\mathcal{G}, \{q_i\})$
```

Figure 9: Pseudocode to compute a fast schedule.

among the tables that correspond to this node. Details are laborious but Figure 8 (right) illustrates an example where the clique t_2, t_3, t_4 can be thought of as a single virtual node. The complexity of the modified junction tree algorithm increases with the size of the largest clique, and hence, we fall back to a modified version of an approximate inference algorithm (CycleScheduler, line#15 in Figure 9) when cliques are large [67]. In general, more complex join graphs require more epochs and exchange more diPs before partition subsets stabilize.

4. USING DATA STATISTICS TO BUILD diPs

Data statistics play a key role in constructing data-induced predicates; recall that the three steps in Equations 1– 3 rely on data statistics; the statistics determine the cost of these operations as well as their effectiveness. 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 [12] 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. Typically, zone-maps store hashes for strings, and so equality check is also a logical equality, but regular expressions are not supported.

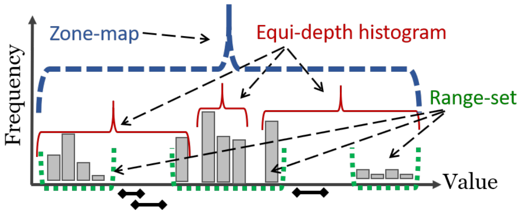


Figure 10: Illustrating the difference between range-sets, zone-maps and equi-depth histogram; both histograms and range-sets have three buckets. The predicates shown in black dumbels below the axes will be false positives for all stats but the range-set.

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

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 can lead to sizable I/O savings (see Figures 11c and 14).

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 [45] 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 [27, 49, 45], are expensive to maintain [49], and the frequency information in histograms, while very useful for other purposes, is a waste of space here because predicate satisfaction and diP construction only check for existence of values.

Bloom filters record set membership [36]. However, we found them to be less useful for our purpose because the partition sizes used in practical distributed storage systems (e.g., $\sim$ 100MBs of data [37, 88]) result in millions of distinct values per column in each partition, especially for join columns which are keys. Recording such large sets requires large space or will have a high false positive rate; for e.g., recording a million distinct values in a 1KB bloom filter has a 99.62% false positive rate [36] and almost no data skipping.

Alternatives such as the count-min [46] and AMS [33] 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 can be 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 increases the likelihood of 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 range-sets as well as checking for overlap between two range sets uses $O(n \log n)$ time where n is the size of larger rangeset; proof is in §**??**. Our results will show that small numbers of ranges (e.g., 4 or 20) lead to large improvements over zone-maps (Figure 17).

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 Eqns. 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 rows in that partition change. 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.

More specifically, the operations over statistics (Eqns. 1–3) are updated as shown below, where t_i^x is true if and only if the x'th partition of the i'th table is tainted.

$$\forall \text{ table } i, \text{ partition } x, \qquad q_i^x \leftarrow t_i^x \vee \mathsf{Satisfy}(p_i, x), \tag{4}$$

$$\forall \text{ tables } i, j, \text{ if } \forall x, t_i^x = 0, \qquad d_{i \rightarrow j} \leftarrow \mathsf{DataPred}(q_i, p_{ij}), \tag{5}$$

$$\forall \text{ table } j, \text{ partition } x, \quad q_j^x \leftarrow t_j^x \vee q_j^x \prod_{i \neq j} \mathsf{Satisfy}(d_{i \rightarrow j}, x). \tag{6}$$

Taint bits can be maintained at transactional speeds and can be extremely effective in some cases, e.g., when updates are mostly in tables which 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 but with a small increase in update cost. See results in §5.3. Taint bits do not suffice, i.e., they will sacrifice I/O savings, if the tables that have query predicates (and which will originate data-reductive diPs) are updateable; for such cases, we propose a different method below that *grows* the data statistics.

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 but otherwise either grow an existing range to contain the new value or collapse two existing 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*.

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	$\{[3,6],[10,27],[52,52]\}$

Table 4: Greedily growing a *range-set* in the presence of updates.

Our results will show that such an update is fast (Table 7), and the reduction in I/O savings is small 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 (Figure 15a).

We also have some hardness results regarding the non-existence of an optimal data statistic for diPs in §**??**, i.e., a statistic cannot simultaneously be small in size, mergeable and avoid false positives on general data distributions. Optimal updates to a range-set also appear hard; that is, as data arrives in a streaming fashion, approximating the optimal total width of a range-set to within a constant factor requires memory that is linear in the number of data values (see §11).

5. EVALUATION

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

- Do data-induced predicates offer sizable gains for a 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.

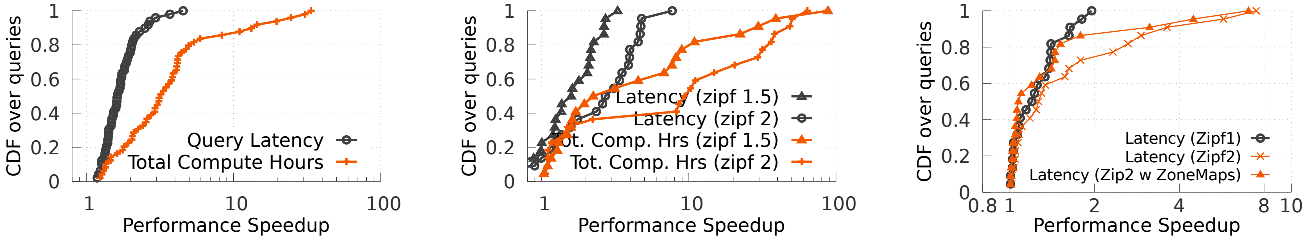
5.1 Methodology

Queries: We report results on TPC-H [80], TPC-DS [25] and the join order benchmark (JOB) [58]. 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 [26] 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 [58].

[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 §**??** for proof of optimality.

[8] $1 \ldots 40, 90 \ldots 99$ from TPC-DS and $([1-9]|10)*$ from JOB



(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

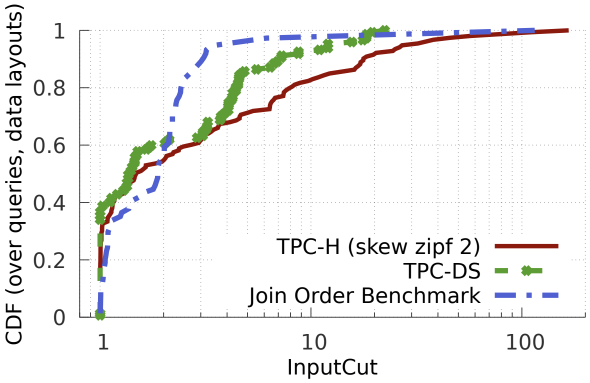
Figure 11: Change in query performance from using data-induced predicates. The figures show CDFs of speedups for different benchmarks, on different platforms for the tuned data layout (see §5.1). The benefits are wide-spread, i.e., almost all queries improve; in some cases, the improvements can be substantial.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

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, [42, 88] and roughly 1M rows per columnstore segment in SQL Server [7].

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 [42, 88] and SQL Server 2016. Both systems use cost-based query optimizers [50]. 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 [18], predicate pushdown to eliminate partitions [6] and magic-set rewrites [47].

Comparisons: In addition to the above production baselines, we compare against several alternatives. By DenormView, we refer to a technique that avoid 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 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 [77]. We apply FineBlock on the above denormalized view.

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



Benchmark	INPUTCUT at percentile				
	50th	75th	90th	95th	100th
TPC-H	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: The INPUTCUT from diPs for different benchmarks; each CDF is over the queries listed in §5.1 and over multiple layouts of the datasets. The table below reads out values at various percentiles; observe that in all the benchmarks (JOB, TPC-DS and TPC-H).

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 How much do derived predicates help?

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 repartitioning for joins and enhances data skipping [19, 23, 48]. 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

[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 §12 .

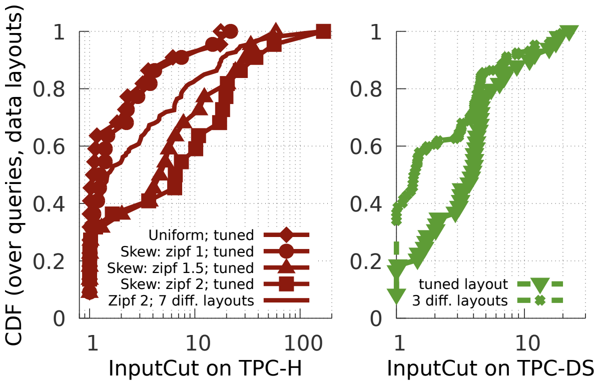


Figure 13: How input skew and data layouts affect the usefulness of diPs; see §5.2.

	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

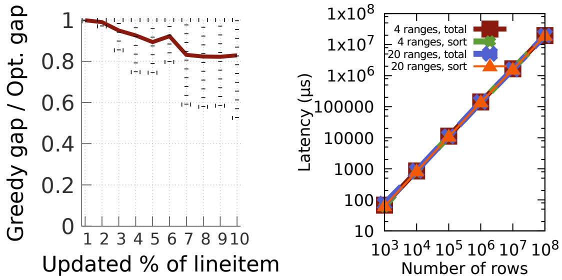
Table 5: Showing different percentiles of the additional latency to derive diPs compared to the baseline QO latency; see §5.2.

	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	$\sim$ 2MB	$\sim$ 35MB	$\sim$ 30KB
# Partitions	$\sim 10^3$	$\sim 4 * 10^4$	$\sim$ 200

Table 6: Additional results for experiments in Figure 11, Figure 12 and Figure 13. The table shows data from our SCOPE cluster experiments for the range-set statistic.

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

Table 7: The time to greedily update range-sets of various sizes measured on a desktop.



(a) Greedy updates.

(b) Construction latency

Figure 15: (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.

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 [44]. We see that almost all queries improve. SCOPE clusters are shared by 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 [26]. 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 [6, 18, 47]; these figures show that diPs offer sizable gains for a sizable fraction of benchmark queries on top of such optimizations.

Figure 12 considers many different layouts, and Figure 13 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. About 20% of the cases receive substantial INPUTCUT, 2.5×, 4.5× and 8× for JOB, TPC-DS and TPC-H respectively. The fraction of cases that receive at least an order of magnitude speed-up (x=10) is 2%, 5% and 19%respectively. Figure 13 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 14 breaks-down the gains for each query in TPC-H when 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, increasing the query optimization duration (as it has to determine which partitions can be skipped), and maintaining statistics under data updates. In big-data clusters, queries are read-only and datasets are bulk appended; so statistic construction and maintenance are less impactful relative to the persistent storage space and QO overhead. Table 5 shows that the 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 fact tables which takes a long time because such diPs have many clauses and, on the receiving table, are evaluated on many partitions. We note that our derivation of diPs is a research 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 any optimizations. We believe that the additional QO time can be substantially lowered. Table 6 shows the size of the

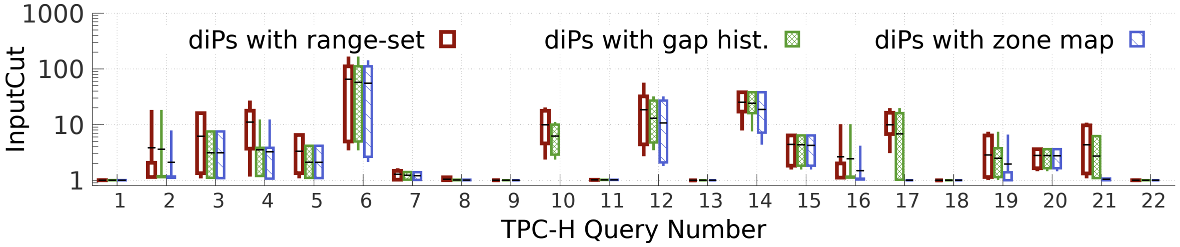


Figure 14: The figure shows the INPUTCUT values when using different stats to construct diPs (for TPC-H with zipf 2 skew). Candlesticks show variation across seven different data layouts including the tuned layout; the rectangle goes from 25th to 75th percentile, the whiskers go from min to max and the thin black dash indicates the average value. Zone-maps do quite well and range-sets are a sizable improvement.

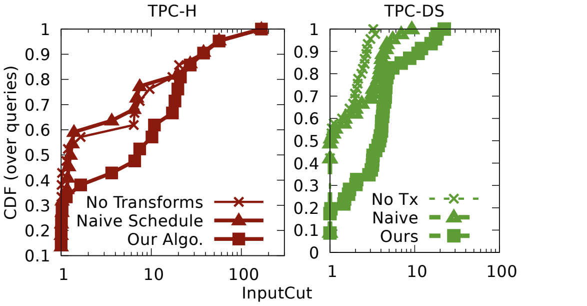


Figure 16: The figure shows how INPUTCUT varies when different methods are used to derive diPs; we compare our algorithm from §3.4 with a naïve algorithm that constructs the same number of diPs. (Results are for TPC-H skewed with zipf 2 and TPC-DS in the tuned layout; other cases behave similarly.)

range-set statistic which can be thought off as roughly 20 "rows" per partition; a partition is 100MB of data in SCOPE clusters and 1M rows in a columnstore segment in SQL server [7]. Hence, the space overhead for range-sets is $\sim$ 0.002%. The space overhead for zone-maps will be 10× smaller because they only record the max and min value per column, i.e., 2 "rows" per partition. TPC-DS and JOB have more tables and more columns, but the 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. Of the 15/22 TPC-H queries in Figure 14 where diPs deliver sizable gains, 5 queries remain unaffected; specifically {q2, q14, q15, q16, q17, q19}. For these queries, diPs lead to large I/O savings in spite of updates. Since updates in TPC-H target the two largest tables, `lineitem` and `orders`, both relations become tainted and diPs cannot flow *from* either of these relations, and so any query that requires a diP out of these tables loses INPUTCUT due to taints. As noted in §4, taints are better suited when updates target smaller dimension tables.

Range set construction time: Figure 15b shows the latency to construct range-sets. We see that computing larger range-sets (e.g., 4 ranges vs. 20) 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). The results here use std::sort method from Microsoft Visual C++. Note that range-sets can be constructed during data ingestion and can be parallelized across partitions and columns; construction can also piggy-back on the first query to scan an input.

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 7 shows that range-sets can be updated in tens of nanoseconds using one core on a desktop; thus, the anticipated slowdown to a transaction system is negligible. Figure 15a 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 $\geq$ 80% of that of the optimal range-set when up to 10% of rows in the `lineitem` table are updated.

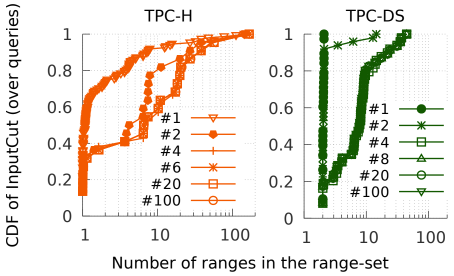


Figure 17: The figure shows how INPUTCUT varies with the numbers of ranges used in the range-set statistic. (Results are for TPC-H skewed with zipf 2 and TPC-DS in the tuned layout; other cases behave similarly.)

5.4 Understanding why diPs help

Comparing different methods to construct diPs: Figure 16 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 17 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 fundamental limit to gains based on the joint distribution of join and predicate columns. Hence, we believe that achieving more I/O skipping beyond diPs with a small number of ranges will require substantially larger statistics and/or more complex techniques.

To understand the result in Figure 14 further, we assess how often the conditions C1–C3 noted in §2.1 for when diPs yield large gains,

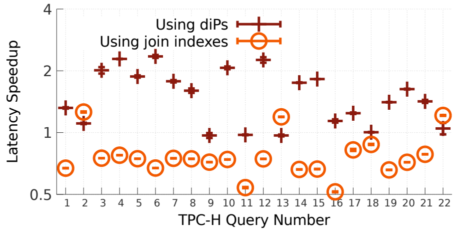


Figure 18: Comparing diPs with join indices on SQL server.

hold for TPC-H queries.

- 15/22 queries receive large I/O savings; namely {**q2**, **q3**, **q4**, **q5**, q6, q7, **q10**, **q12**, q14, q15, **q16**, **q17**, **q19**, q20, **q21**}.
- For the queries shown in **red** above, diPs magnify the gains from predicate pushdown, predicates on small relations can now eliminate many partitions on the large relations.
- Among the remaining queries:
  - {q1, q6} have no joins; so diPs do not offer additional value.
  - {q7, q14, q15, q20} have selective predicates only on the largest table and so diPs only offer modest gains over predicate pushdown.
  - {q9, q13, q18, q22} have no predicates or predicates with low selectivity.
  - {q8, q11} violate C1 on all seven layouts, i.e., rows picked by the predicate are spread over many partitions. {q2, q5, q7, q16} violate C1 on most but not all of the layouts; hence their gains from diPs vary substantially across layouts.
  - {q7} violates C2 and C3 on all seven layouts; {q16, q19, q17, q20, q21} violate C2 and C3 in some but not all of the layouts which translates to high variance in gains across layouts.

Table 8 offers additional detail with more detailed conditions than those mentioned in §2.1.

5.5 Comparing with alternatives

Join Indexes: Figure 18 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 [5] 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`). Indexes are not supported on columnstores; so we use rowstores for just this experiment. The figure shows that join indexes lead to worse query latency than using default SQL server without indexes; we believe this is because: (1) the predicate selectivity in TPC-H queries is not small enough to benefit from an index seek leading most plans to use a clustered index scan to read the input relations, 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 19 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 (view statement is in §13). This view in columnar (rowstore) format occupies 2.7× (6.2×) more storage space than all of the other tables combined. Queries that can use this view are un-hindered 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 each table is in a columnar layout. Figure 20 (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) as expected. Materialized views speed-up queries, in general, only when the view statements have selections or aggregations that reduce the size of the view [31]. For the case here, there are no common predicates or aggregations across the 16/22 queries in TPC-H.

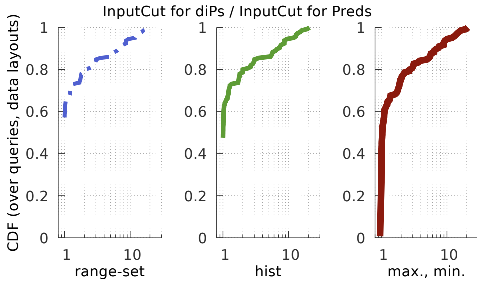


Figure 19: Comparing diPs versus using the same stats to only skip partitions on individual tables.

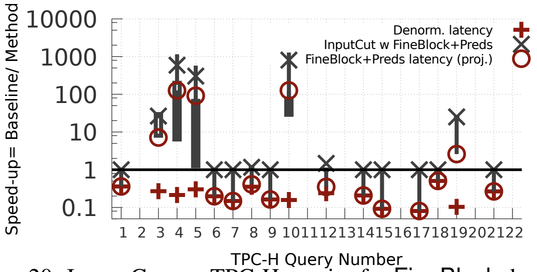


Figure 20: InputCut on TPC-H queries for FineBlock; box plots show results for different predicates.

diPs vs. clustering rows by predicates: A recent research proposal [77] clusters rows in the above view to maximize data skipping. Training over a set of predicates, [77] learns a clustering scheme that intends to skip data for unseen predicates. Figure 20 shows with x symbols the average InputCut obtained as a result of such clustering; the candlesticks around the x symbol show the min, 25th percentile, 75th percentile and max InputCut for different query predicates. We see that most queries receive no InputCut (x 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 20 also shows with circle symbols the average query latency when using this clustering. 5/22 queries improve (fastest query is ∼ 100× faster) while 11/22 queries regress (slowest is 10× slower). Hence, the practical value of this scheme is unclear.

We also note the rather large overheads to create and maintain indexes and views [30, 68] and to learn clusterings [77]. These schemes also require foreknowledge of queries and offer gains only for future queries that are similar [31, 32, 77]. In contrast, diPs only use small and easily maintainable data statistics, require no apriori knowledge of queries and we show can offer sizable improvements for ad-hoc and complex queries.

6. DISCUSSION

Other uses of diPs: It is possible to use diPs for purposes other than eliminating partitions during query optimization. For example, during query execution, a diP can be used as a SARG-able predicate on an index [71]. A diP can also be sent to a remote store [20] such that only data satisfying the predicate is fetched from the store; doing so helps when storage is disaggregated because a common bottleneck in such systems is the path between the compute and store. In-memory engines can also benefit from executing diPs within the query; by doing so, even though the initial I/O remains the same, joins can speed up because they process less data, and executing diPs can be more efficient than constructing bloom filters or bitmaps for semijoin optimizations [18].

Compressed or encrypted stores: Since computing and applying diPs only uses partition statistics, their gains are not impacted if the underlying stores are compressed or encrypted [57].

Compaction of diPs: In some cases, a diP can have too many clauses. For example, when using range-sets with n_r ranges, if n_p partitions match a predicate, then the diP can be a disjunction of up to $n_r * n_p$ clauses. To bound the cost of evaluating diPs, we limit each diP to have no more range clauses than a specified threshold; optimal compaction has $O(n_r \log n_r)$ complexity.

Convergence under compaction: The convergence claims in §3.4 and §10 do not hold when diPs are compacted as above because compaction is lossy. For the sake of simplicity, our implementation uses the schedules described in §3.4 and repeats them until no more partitions are eliminated. In practice, we find that the additional diP computations needed are negligible.

Plan caches: Note that the diPs for a query depend on the data; if the underlying data changes, then the diPs may have to be recalculated and hence, using diPs reduces the value of plan caches. We note that cached plans can be reused as long as datasets do not change; most queries in big-data clusters are read-only which helps this case. Moreover, cached plans are not useful if datasets change in a substantial way (e.g., the size of some input doubles or some input layout is changed).

7. 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, diPs are built on-the-fly in the QO and can skip partitions of multiple joining relations, diPs 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 [9, 39, 53, 55]. Inferring such data properties is a sizable cost (e.g., [53] 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 [9] uses the fact that the `l_shipdate` attribute of `lineitem` is between 0 to 90 days larger than `o_orderdate` from `orders` [80] to convert predicates on `l_shipdate` to predicates on `o_orderdate` and vice versa. Others discover similar constraints more broadly [39, 55]. 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 data layouts are clustered on time (e.g., log data) or when tables are partitioned on join columns [48].

Prior work that moves predicates around relies on column equivalence and magic-set style reasoning [47, 59, 61, 72, 81, 83]. Both SCOPE clusters and SQL server implement such optimizations, and as we saw in §5, diPs offer gains on top of these baselines. Column equivalence does not help much when predicate columns do not exist in joining relations. Magic set transformations are shown to only help 2/22 of the TPC-H queries and only when predicates are highly selective [72]. Inferring new predicates that are induced by data statistics allows us to have wider appeal.

Auxiliary data structures such as views [24], join indices [21], join bitmap indexes [4], succinct tries [87], column sketches [51], and partial histograms [84] can also help speed-up queries. Join zone maps [13] 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. Complex queries, especially those that join on foreign-keys, e.g., `store_sales` and `store_returns` in TPC-DS are joined in six different ways, can require maintaining many different structures. diPs can be thought off as a complementary approach that helps with or without such auxiliary structures.

While data-induced predicates are similar to the implied integrity constraints used by [63], there are some key differences and additional contributions. (1) [63] only exchanges constraints between a pair of relations, we offer a general method which exchanges diPs between multiple relations, handles cyclic joins and supports queries having group-by's, union's and other operations. (2) [63] uses zone maps and two bucket histograms; we offer a new statistic (range-set) that performs better. (3) [63] shows no query performance improvements; we show speed-ups in both a big-data cluster and a DBMS. (4) [63] offers no results in the presence of data updates; we design and evaluate two maintenance techniques that can be added to transactional systems and show that the diPs offer gains even when large fractions of datasets are updated.

While a query executes, sideways information passing (SIP) from one subexpression to a joining subexpression can prune the data-in-flight and speed up the query [35, 54, 61, 66, 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 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 easily extend to the case of multiple joins 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[77, 78, 60, 64, 74, 60, 74, 86, 78, 64, 41, 40, 52, 70]; they co-locate data that is accessed together or build correlated indices. Some use denormalization to avoid joins [77, 86]. In contrast, diPs require no changes to the data layout and no foreknowledge of queries.

8. 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.

9. 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] Bitmap join indexes in oracle. `https://bit.ly/2TLBBTF`.

[5] Clustered and nonclustered indexes described. `https://bit.ly/2Drdb9o`.

[6] Columnstore index performance: Rowgroup elimination. `https://bit.ly/2VFpljV`.

[7] Columnstore indexes described. `https://bit.ly/2F7LZuI`.

[8] Data skipping index in spark. `https://bit.ly/2qONacb`.

[9] Date correlation optimzation in sql server 2005 & 2008. `https://bit.ly/2VodSVN`.

[10] Imdb datasets. `https://imdb.to/2S3BzSF`.

[11] Join order benchmark. `https://bit.ly/2tTRyIb`.

[12] Oracle database guide: Using zone maps. `https://bit.ly/2qMeO9E`.

[13] Oracle: Using zone maps. `https://bit.ly/2vsUWKK`.

[14] Parquet thrift format. `https://bit.ly/2vm6D5U`.

[15] Presto: Repartitioned and replicated joins. `https://bit.ly/2JauYll`.

[16] Processing petabytes of data in seconds with databricks delta. `https://bit.ly/2Pryf2E`.

[17] Query 1a in job. `https://bit.ly/2Fomtmx`.

[18] Query execution bitmap filters. `https://bit.ly/2NJzzgF`.

[19] Redshift: Choosing the best sort key. `https://amzn.to/2AmYbXh`.

[20] S3 sequential scan. `https://amzn.to/2PHd38g`.

[21] Teradata: Join index. `https://bit.ly/2FbalDT`.

[22] Tpc-ds query #35. `https://bit.ly/2U0rIk6`.

[23] Vertica: Choosing sort order: Best practices. `https://bit.ly/2yrvPtG`.

[24] Views in sql server. `https://bit.ly/2CnbmIo`.

[25] TPC-DS Benchmark. `http://bit.ly/1J6uDap`, 2012.

[26] Program for tpc-h data generation with skew. `https://bit.ly/2wvdNVo`, 2016.

[27] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. *SIGMOD Rec.*, 1999.

[28] P. K. Agarwal et al. Mergeable summaries. *TODS*, 2013.

[29] S. Agarwal et al. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[30] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient view maintenance at data warehouses. In *ACM SIGMOD Record*, 1997.

[31] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. *VLDB*, 2000.

[32] S. Agrawal et al. Database tuning advisor for microsoft sql server 2005. *VLDB*, 2004.

[33] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *STOC*, 1999.

[34] M. Armbrust et al. Spark sql: Relational data processing in spark. In *SIGMOD*, 2015.

[35] F. Bancilhon et al. Magic sets and other strange ways to implement logic programs. In *SIGMOD*, 1985.

[36] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 1970.

[37] D. Borthakur et al. Hdfs architecture guide. *Hadoop Apache Project*, 2008.

[38] D. Borthakur et al. Apache hadoop goes realtime at facebook. In *SIGMOD*, 2011.

[39] P. G. Brown and P. J. Haas. Bhunt: Automatic discovery of fuzzy algebraic constraints in relational data. In *VLDB*, 2003.

[40] M. Brucato, A. Abouzied, and A. Meliou. A scalable execution engine for package queries. *SIGMOD Rec.*, 2017.

[41] L. Cao and E. A. Rundensteiner. High performance stream query processing with correlation-aware partitioning. *VLDB*, 2013.

[42] R. Chaiken et al. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *VLDB*, 2008.

[43] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 2012.

[44] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD*, 2015.

[45] 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.

[46] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 2005.

[47] M. Elhemali, C. A. Galindo-Legaria, T. Grabs, and M. M. Joshi. Execution strategies for sql subqueries. In *SIGMOD*, 2007.

[48] M. Y. Eltabakh et al. Cohadoop: Flexible data placement and its exploitation in hadoop. In *VLDB*, 2011.

[49] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. In *VLDB*, 1997.

[50] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 1995.

[51] B. Hentschel, M. S. Kester, and S. Idreos. Column sketches: A scan accelerator for rapid and robust predicate evaluation. In *SIGMOD*, 2018.

[52] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, 2007.

[53] I. Ilyas et al. Cords: Automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, 2004.

[54] Z. G. Ives and N. E. Taylor. Sideways information passing for push-style query processing. In *ICDE*, 2008.

[55] H. Kimura et al. Correlation maps: a compressed access method for exploiting soft functional dependencies. In *VLDB*, 2009.

[56] A. Lamb et al. The vertica analytic database: C-store 7 years later. *VLDB*, 2012.

[57] H. Lang et al. Data blocks: Hybrid oltp and olap on compressed storage using both vectorization and compilation. In *SIGMOD*, 2016.

[58] V. Leis et al. How good are query optimizers, really? In *VLDB*, 2015.

[59] A. Y. Levy, I. S. Mumick, and Y. Sagiv. Query optimization by predicate move-around. In *VLDB*, 1994.

[60] Y. Lu, A. Shanbhag, A. Jindal, and S. Madden. AdaptDB: Adaptive partitioning for distributed joins. In *VLDB*, 2017.

[61] I. S. Mumick and H. Pirahesh. Implementation of magic-sets in a relational database system. In *SIGMOD Record*, 1994.

[62] A. Nanda. Oracle exadata: Smart scans meet storage indexes. `http://bit.ly/2ha7C5u`, 2011.

[63] A. Nica et al. Statisticum: Data Statistics Management in SAP HANA. In *VLDB*, 2017.

[64] M. Olma et al. Slalom: Coasting through raw data via adaptive partitioning and indexing. *VLDB*, 2017.

[65] C. Olston et al. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, 2008.

[66] J. M. Patel et al. Quickstep: A data platform based on the scaling-up approach. In *VLDB*, 2018.

[67] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[68] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *SIGMOD Record*, 1996.

[69] M. Saglam and G. Tardos. On the communication complexity of sparse set disjointness and exists-equal problems. In *FOCS*, 2013.

[70] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The uncracked pieces in database cracking. *VLDB*, 2013.

[71] P. G. Selinger et al. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[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. *VLDB*, 2016.

[75] L. Shrinivas et al. Materialization strategies in the vertica analytic database: Lessons learned. In *ICDE*, 2013.

[76] D. Ślęzak et al. Brighthouse: An analytic data warehouse for ad-hoc queries. *VLDB*, 2008.

[77] L. Sun et al. Fine-grained partitioning for aggressive data skipping. In *SIGMOD*, 2014.

[78] L. Sun, M. J. Franklin, J. Wang, and E. Wu. Skipping-oriented partitioning for columnar layouts. In *VLDB*, 2017.

[79] A. Thusoo et al. Hive- a warehousing solution over a map-reduce framework. In *VLDB*, 2009.

[80] TPC-H Benchmark. http://www.tpc.org/tpch.

[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. *VLDB*, 2016.

[85] M. a. Zaharia. 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. *VLDB J.*, 2012.

[89] J. Zhou, P.-A. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the scope optimizer. In *ICDE*, 2010.

10. CONVERGENCE PROOF

We will now prove that the scheduling algorithm presented in §3.4 produces convergent schedules. This proof makes no assumptions on the partition statistics used, as long as the statistics meet the criteria listed in §2, that is, they can identify satisfying partitions and are mergeable [28]. The proof assumes that merging statistics is not lossyl i.e., the merged stat corresponding to a set of stats has the exact same information.

Let's consider an acyclic join graph, $\mathcal{G}$. The schedule from §3.4 passes data-induced predicates from the leaves of the join tree up to the root and then back down.

One join: Suppose we have a single join between two relations, table t and table r, as shown in the Figure 8(left). The schedule for this join graph first applies local predicates to pick partitions that satisfy the predicate on each table, say $q_{t,(0)}$ and $q_{r,(0)}$ (the (0) indicates these vectors are values before the first epoch). It then exchanges diPs $d_{t\rightarrow r,(1)}$ and $d_{r\rightarrow t,(1)}$ between the tables t and r (the (1) indicates these are diPs from epoch (1)). Lastly, each table updates their partition subsets based on these diPs to get $q_{t,(1)}$ and $q_{r,(1)}$ respectively.

To show that $q_{t,(1)}$ and $q_{r,(1)}$ are converged, suppose the contrarian case that some partition in either table is eliminated by another exchange of data-induced predicates. Without loss of generality, suppose partition x in table t is newly eliminated in the second epoch; that is, $q^x_{t,(2)} = 0$ but $q^x_{t,(1)} = 1$. Let us think about the join column values that are present in partition x.

On the one hand, partition x must have satisfied the local predicate on table t in epoch (0).

On the other hand, since x is newly eliminated in epoch (2), there must have been some change in the diP from table r to cause this; that is, some partition subset S in table r must have been eliminated at the end of epoch (1), and the join column values in x must only overlap with the partitions in S in order for the elimination of partitions in S on table r to cause the elimination of x in table t. That is, for the join column values, $x \subset S$. Furthermore, because the partitions in S were eliminated at epoch (1), none of their join column values satisfy the derived local predicate on table t, i.e.

$d_{t \to r,(1)}$. Since $x \subset S$, none of the values in x satisfy the local predicate on t.

Chain with three tables and two joins: Consider a join graph r – s – t with three tables. The algorithm in §3.4 has the following steps. First, all tables apply local predicates if any. In epoch (1), the diPs $d_{r \to s,(1)}$ and $d_{t \to s,(1)}$ are computed and used by table s to update its partition subset. In epoch (2), the diPs $d_{s \to r,(2)}$ and $d_{s \to t,(2)}$ are computed, and tables r and t update their partition subsets.

To show that the partition subsets have now converged, note that there are four possible diPs that can be computed on this join graph, and we will show that none of these diPs can change a partition subset.

To see why the diP $d_{r \to s,(3)}$, apply the "one join" case above with the "local predicate" on tables s and r being $p_s \wedge d_{t \to s,(1)}$ and p_r, respectively. Start with with r applying its local predicate and deriving $d_{r \to s,(1)}$. Then, the same contradiction above will apply here as well.

The same argument holds for the remaining three diPs of $d_{s \to r,(3)}$, $d_{s \to t,(3)}$, and $d_{t \to s,(3)}$. The "local predicate" on table s is always $p_s \wedge d_{j \to s,(1)}$ where j is either t or r. The "local predicate" for table t and r is p_t and p_s, respectively.

Arbitrary Tree: We will prove this case inductively by starting at the root of the tree. Consider the chain containing the left child of the root, the root and the right child of the root. Applying the logic above, we can show that none of the four diPs on these edges will change partition subsets on the tables involved here. Given this holds we can now apply the same logic on the two subtrees one having the left-child as the root and another having the right-child as the root.

11. HANDLING UPDATES TO DATASETS

The primary use-case for diPs is data warehouses and big-data clusters where datasets are read-only or are appended to in large batches. In this cases, statistics can be constructed on newly arriving batches before making the data available to queries. We note that this is a widely prevalent use-case; it occurs in all large data-parallel clusters today.

Extending the case above, we discuss using diPs when the datasets can be updated. That is, rows can be deleted, new rows can be added or one or more attributes in a row can change. The challenge in handling updates is that if the data statistics are not modified in accordance with the updates to data, the statistics can give rise to incorrect data-induced predicates (which may prune partitions that should not be pruned) and therefore lead to incorrect query answers. We have already discussed two approaches in §4– using a taint bit per partition to identify partitions that have changed data and greedily growing the range-set statistic to cover all new values. Here we add some comments.

It is easy to see that the cost of maintaining one taint bit per partition is trivial. Updates to different rangesets, e.g., the range-sets of different columns and different partitions, are trivially parallelizable. Finer granularity taint bits, e.g., one taint bit per column and per partition as opposed to just a single taint bit for all columns in a partition can offer greater data skipping value (because diPs can originate at a dataset as long as the join columns related to that diP are untainted even if the other columns are tainted). In this way, finer granularity taints can trade-off a small increase in maintenance cost for a possibly large improvement in gains from data skipping.

Is there an optimal streaming update procedure for rangeset? That is, in a streaming manner as the dataset evolves (with updates, insertions and deletions), can the corresponding rangeset be updated optimally? Recall that the best rangeset has the largest total *gap* between the ranges. Unfortunately, the answer is no. Consider a simple scenario: building a range-set of size 2 with only insertions; assume that the stream has a total size of n values, and the update process is restricted to store no more than $n/4$ values. Since the range-set is of size 2, the problem devolves to identifying the largest gap between the values. The following counter-example achieves a competitive ratio of nearly 3; that is the gap identified by the online procedure is 3× smaller than optimal. (1) Let the first $(n/4) + 2$ rows be evenly distributed across the value space from minimum to maximum value. Since the online process can only store $n/4$ gap values, the $(n/4) + 2$'th value will create the $(n/4) + 1$'th gap and cannot be stored. So, the online process has to *forget* one of these $(n/4) + 1$ gaps. (2) Use the remaining values in the stream to evenly break up each of the $n/4$ gaps that the online process remembers, making whichever gap was forgotten first to be the largest gap overall and ensuring that no remembered gap is larger than 3× the forgotten gap value. We can ensure this because $(3n/4) - 2$ values remain to break up the $(n/4)$ gaps that are remembered. A more complex construction can lead to an even larger competitive ratio. Streaming procedures often cannot store $n/4$ values; they typically have a constant or $\log n$ memory budget, and along the lines of the intuition above, we can show that with a constant budget k, the competitive ratio can be as large as $1 + \lceil \frac{n-k+2}{k} \rceil$. Thus, we eschew pursuit of an optimal update procedure and rely on a greedy update process that is always quick and useful in practice.

12. TUNED DATA LAYOUTS

The tuned data layouts that we use in our evaluation laid out the tables in the following manner.

12.1 TPC-H

The table lineitem is hash-clustered on l_shipdate and each cluster is internally ordered by l_orderkey. The table orders is hash-clustered on o_orderdate and each cluster is internally ordered by o_orderkey. The table partsupp is sorted by ps_partkey. All other tables are sorted on their primary key.

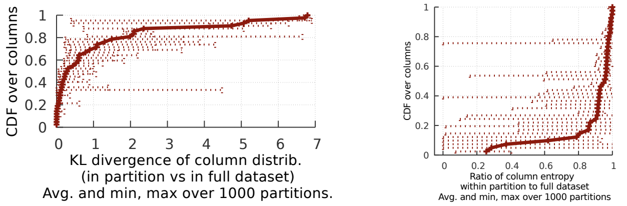
12.2 TPC-DS

The tables store_sales, store_returns, catalog_sales, catalog_returns, web_sales and web_returns are hash clustered on date columns, specifically ss_sold_date_sk, sr_returned_date_sk, cs_sold_date_sk, cr_returned_date_sk, ws_sold_date_sk and wr_returned_date_sk respectively. All other tables are hash clustered on their primary keys.

13. DENORMALIZATION OF TPC-H

The following materialized view (or denormalized table) can support 16 out of 22 queries in the TPC-H benchmark [80]; specifically, queries $\{2, 11, 13, 16, 20, 22\}$ cannot be answered using just this view because those queries require information that is absent in the view.

CREATETABLE denorm **AS**
SELECT lineitem.*, customer.*, orders.*, part.*, partsupp.*, supplier.*, n1.*, n2.*, r1.*, r2.*
FROM lineitem **JOIN** orders **ON** o_orderkey = l_orderkey
JOIN partsupp **ON** ps_partkey = l_partkey **AND** ps_suppkey = l_suppkey
JOIN part **ON** p_partkey = ps_partkey



(a) KL divergence between per partition and whole dataset distributions of column values.

(b) Entropy within partition vs. entropy on full dataset.

Figure 21: Analysis of logs from production

JOIN supplier **ON** s_suppkey = ps_suppkey
JOIN customer **ON** c_custkey = o_custkey
JOIN nation **AS** n1 **ON** n1.n_nationkey = c_nationkey
JOIN nation **AS** n2 **ON** n2.n_nationkey = s_nationkey
JOIN region **AS** r1 **ON** r1.r_regionkey = n1.n_regionkey
JOIN region **AS** r2 **ON** r2.r_regionkey = n2.n_regionkey

14. ADDITIONAL RESULTS

As additional motivation for diPs, we analyzed the *in situ* layouts of crawled web snapshots, click logs and server logs to understand how predicate and join columns are distributed. We use 1000 partitions for each dataset; each partition is roughly 100MB of data in our production clusters. We compute the global and per-partition histograms of each column which we denote as $\texttt{Hist}(c)$ and $\texttt{Hist}(c, p)$ respectively for column c and partition p. Figure 21a shows a CDF (over columns) of the distance (specifically: KL divergence value) between these two histograms; the larger the distance, the further the distribution of column values in a partition is from the overall distribution. The line in the figure joins the average, and the errorbars are the min and max values over all partitions. Note that many columns and many partitions have nearly the highest possible distance.[10] As further evidence, Figure 21b shows the ratio of the entropy of a column within a partition to its entropy across the entire dataset. The line is a CDF over columns of the average entropy ratio across partitions and errorbars denote the min and the max. An entropy ratio close to 1 indicates that the column values in partition have the same entropy as they do over all of the dataset. However, as the figure shows, several columns have much smaller entropy on many partitions indicating clustering. We conclude that practical datasets exhibit the behaviors mentioned above where deriving predicates can lead to sizable data skipping.

14.1 When will diPs give large gains?

Following up on the description in §5.2, Table 8 lists which queries, predicates and data layouts satisfy some detailed conditions required to obtain large gains from data-induced predicates.

14.2 Growth of false positives during construction and use of diPs

In Figure 22, we show how the fraction of rows that match a predicate changes during the construction and use of data-induced predicates. These results are aggregated over all the queries in TPC-H executing on a skewed dataset (zipf 2) over seven different data layouts. The leftmost figure, Figure 22(a), compares the fraction of rows filtered by a predicate with the fraction of partitions containing these rows. Note that the figures are 2d histograms on a logarithmic scale. We see substantial concentration on the $y = 1$ line indicating that many predicates, even those that are selective, may not filter out partitions. Figure 22(b) plots the fraction of partitions picked on a source table versus the fraction of rows in the destination table that match the data-induced predicate constructed on the source table; in a sense, this figure estimates the succinctness of the diP. In this figure, we see even more concentration along the $y = 1$ line indicating that the constructed diPs are not succinct and may match a large number of rows in the destination table. Figure 22(c) plots the fraction of rows matching on the destination table versus the number of partitions on the destination table that contain these rows. Finally, Figure 22(d) shows the cumulative effect of all three steps in the figures on the left. The key takeaway is that, as expected, each step in constructing and applying data-induced predicates adds to false positives; yet, diPs successfully eliminate partitions on the destination relation (note: sizable mass below $y = 0.5$ line in Figure 22(d) which will translate to InputCut= 2.).

14.3 Adaptive partitioning comparison

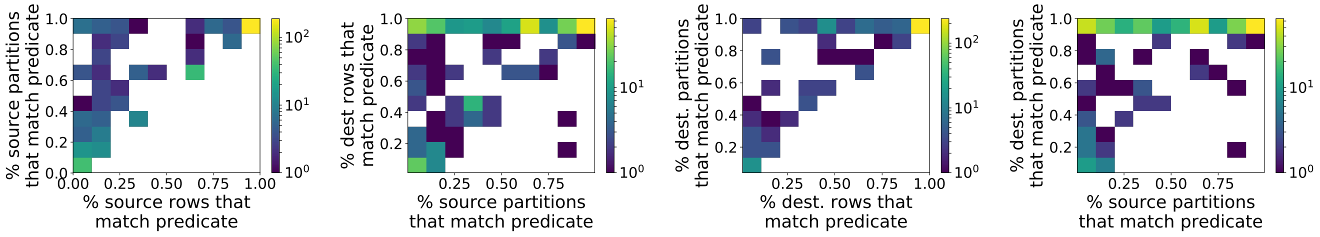
We mention a few additional details regarding our comparison with [77] which learns a clustering scheme over rows of a denormalized relation of TPC-H so as to enhanced data skipping. We had to reimplement the algorithm in [77] because the code shared by the authors was missing some key pieces. We note some key aspects of our implementation, FineBlocks. (1) As described in [77], we first partition rows of the denormalized relation shown in §13 by the month of O_ORDERDATE and then cluster together rows that match (or do not match) the same predicates, excluding date predicates. (2) The authors of [77] have also stated that they rewrote query predicates using hard-coded constraints between the L_SHIPDATE and O_ORDERDATE columns. Such constraints are not available in general across tables; hence, we do not use such rewrites in FineBlock. (3) The FineBlock results use a TPC-H scale factor of 1 because we had trouble scaling to larger dataset sizes; however, we scale down the minimum partition size to create the same number of partitions as in [77] (note: this is $11,000$ partitions). (4) The algorithm in [77] is sensitive to training data and may not work well when the test data is very different from training because the rows are clustered only based on predicates that are available during training. We train FineBlock on 8 query templates with 30 queries each; namely $\{3, 5, 6, 8, 10, 12, 14, 19\}$. We test FineBlock on 16 query templates, 10 queries per template; namely $\{1, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 15, 17, 19, 20, 21\}$. The remaining 6 query templates in TPC-H are not contained in the denormalized relation shown in §13 and hence will receive no benefit from FineBlock+Preds. (5) The time to train the workload-aware partitioning and to re-layout the dataset is sizable (for a 1GB dataset, takes about 2400s, single-threaded, on an x86 linux server with 1TB memory); this process is compute bottlenecked and the time should increase with the number of rows; it should grow much more quickly if the dataset spills from memory. Storing the partitioning metadata of FineBlock requires somewhat less space than the data stats used by diPs; $3500B$ to maintain a dictionary of the predicates used as features for partitioning and roughly $10B$ per partition to store a bit vector of which features are matched by a partition versus about $2000B$ per partition for pdSkip. The time to skip partitions is also roughly similar; about $0.02s$ per query.

Our reimplementation of the algorithm from [77] matches the results in that paper after using the following additional tricks: (a) use domain knowledge to translate predicates on L_SHIPDATE to equivalent predicates on O_ORDERDATE and (b) use many more training

[10] $D_{KL}\left(Hist(c,p)||Hist(c)\right) = \sum_v -Prob_{c,p}(v)\frac{Prob_c(v)}{Prob_{c,p}(v)} \leq \ln(10^3) = 6.91$ because $\texttt{Prob}_{c,p}(v) \leq 10^3 * \texttt{Prob}_c(v)$, $\forall c, p, v$. The last inequality holds because each dataset here has 10^3 partitions and $\sum_p \texttt{Freq}_{c,p}(v) = \texttt{Freq}_c(v)$.

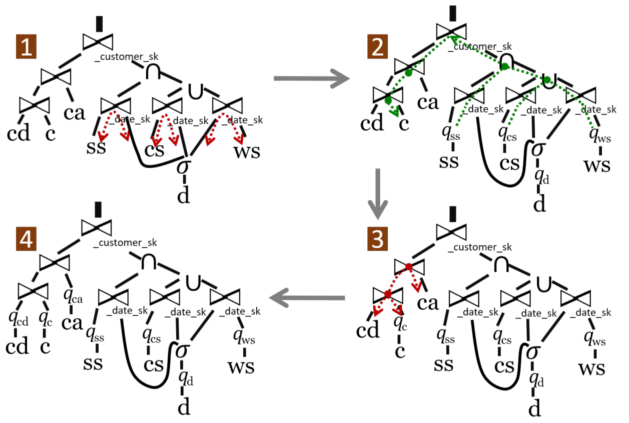
Condition	Queries that **do not** manifest this condition
D1: query has predicates on smaller relation(s)	**q18**
D2: predicates are selective	all preds: (**q1**, **q9**, **q13**, **q22**); some preds: (q2, q3, q4, q5, q6, q7, q8, q10, q11, q12, q14, q15, 16, 17, 19, 20, 21)
D3: rows picked by predicates are concentrated in a few partitions	all layouts: (**q8**, **q11**), most layouts: (q2, q5, q7, q16), some layouts: (q3, q4, q6, q10, q12, q14, q15, q17, q19, q20, q21)
D4: stat can identify skippable partitions	regex: (q2, q9, q13)
D5: join column values belonging to the unskippable partititions of a relation are concentrated in a few partitions of the *joining* relation	all layouts: (**q7**), most layouts: (q16, q19, q20), some layouts: (q17, q21)

Table 8: Analyzing the conditions required to get large gains from deriving predicates over joins. Queries are from TPC-H. Analysis is performed over seven different data layouts when using the range-set statistic; see §5.1 for specifics on setup.



(a) Source: %Row → %Part. (b) Source %Part → Dest %Row (c) Dest: %Row → %Part. (d) Source %Row → Dest %Part

Figure 22: Examining the change in fractions of rows that match a predicate, the fraction of partitions that contain these rows, the fraction of rows in the destination table that match the diPs constructed over matching partitions and finally the fraction of partitions of the destination table that match the diP. Results are for all 197-H queries executing on a skewed dataset (zipf 2) over seven different datalayouts; each predicate and diP contribute one point and the figures show 2-d histograms as heat plots in a logarithmic scale.



% of partitions remaining in...

step	ss	cs	ws	d	c	ca	cd
Initially	100	100	100	100	100	100	100
After local predicates				0.57			
d_1→ss, d_2→cs, d_3→ws	23.9	24.9	25				
{ {cs-d_2} ∪ {ws-d_3}} ∩ {ss-d_1} → c					100		
Final	23.9	24.9	25	0.57	100	100	100

Figure 24: Step-by-step reduction in the fraction of partitions to be read while using data-induced predicates for TPC-DS query 35.

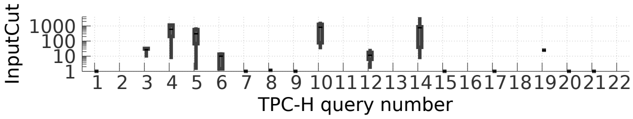
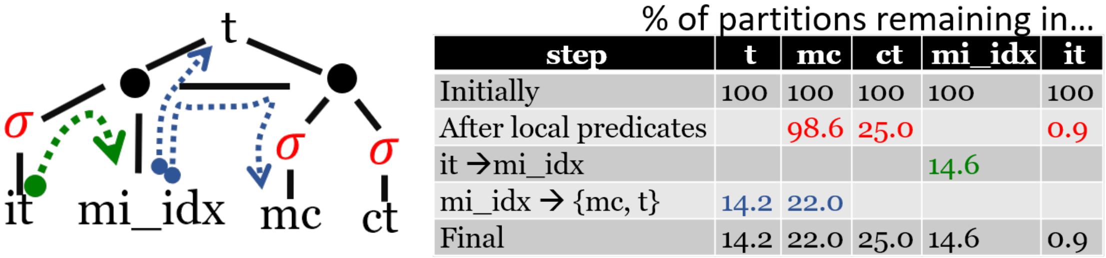


Figure 23: After adding a few changes, which we consider to be impractical, FineBlock can match the results presented in the original paper.

queries [77] such that almost all of the test predicates are available during training. These results are shown in Figure 23.

15. MORE END-TO-END EXAMPLES

Analogous to Figure 5, Figures 24 and 25 illustrate diPs in action for a query in TPC-DS [25] and in JOB [11], respectively. We choose these queries to illustrate how diPs work with complex statements (union operators, nested sql statements in TPC-DS q35 [22]) and cyclical joins (in JOB 1a [17]).



% of partitions remaining in...

step	t	mc	ct	mi_idx	it
Initially	100	100	100	100	100
After local predicates		98.6	25.0		0.9
it →mi_idx				14.6	
mi_idx → {mc, t}	14.2	22.0			
Final	14.2	22.0	25.0	14.6	0.9

Figure 25: For the query 1a in the JOB benchmark, showing how diPs skip input partitions; here t, mc, ct, mi_idx and it correspond to the title, movie_companies, company_type, movie_info_idx, and info_type tables respectively.