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

Srikanth Kandula, Laurel Orr, Surajit Chaudhuri
Microsoft

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

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 [19].

```
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 [55], magic set rewriting [52, 76] or value-based pruning [85]), but predicates over join columns are rare,[1] and these techniques do not apply when the predicates use columns that do not exist in the joining tables.

[1]Over all the queries in TPC-H [28] and TPC-DS [26], there are zero predicates on join columns perhaps because join columns tend to be opaque system-generated identifiers.

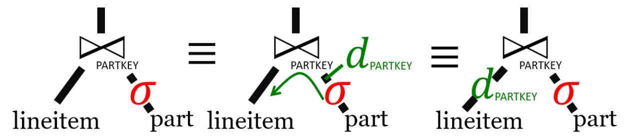


Figure 1: Example illustrating creation and use of a data-induced predicate which only uses the join columns and is a necessary condition to the true predicate, i.e., $\sigma \Rightarrow d_{\texttt{partkey}}$.

Some systems implement a form of sideways information passing over joins [21, 71] during query execution. For example, they may build a bloom filter over the values of the join column `_partkey` in the rows that satisfy the predicate on the `part` table and use this bloom filter to skip rows from the `lineitem` table. Unfortunately, this technique only applies during query execution, does not easily extend to general joins and has high overheads, especially during parallel execution on large datasets because constructing the bloom filter becomes a scheduling barrier delaying the scan of `lineitem` until the bloom filter has been constructed.

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

Our target scenario is big-data systems, e.g., SCOPE [47], Spark[39, 88], Hive [84], F1 [80] or Pig [70] clusters that run SQL-like queries over large datasets; recent reports estimate over a million servers in such clusters [1].

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

Using data statistics, we offer a method that converts predicates on a table to data skipping opportunities on the joining tables at query optimization time. The method, an example of which is shown in Figure 1, begins by using data statistics to eliminate partitions on tables that have predicates. This step is already implemented in some systems [7, 17, 47, 85]. Next, using the data statistics of the partitions that satisfy the local predicates, we construct a new predicate which captures all of the join column values contained in such partitions. This new *data-induced predicate* (diP) is a necessary condition of the actual predicate (i.e., $\sigma \Rightarrow d$) because there may be false-positives; i.e., in the partitions that are included in the diP, not all of the rows may satisfy σ. However, the diP can apply over the joining table because it only uses the join column[2]; in the case of Figure 1, the diP constructed on the `part` table can be applied on the partition statistics of `lineitem` to eliminate partitions. All of these steps happen during query optimization; our QO effectively replaces each table with a partition subset of that table; the reduction in input size often triggers other plan changes (e.g., using broadcast joins which eliminate a partition-shuffle [49]) 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 [48] or using join indexes [5, 25]. The fundamental reason is that these techniques use augmentary data-structures which are costly to maintain; yet, their benefits are limited to narrow classes of queries (e.g., queries that match views, have specific join conditions, or very selective predicates) [36]. 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 [53, 65] or cluster rows that satisfy the same predicates [77, 82]; the former speeds up joins and the latter enhances data skipping. In our big-data clusters, many unstructured datasets remain in the order that they were uploaded to the cluster. The choice of data layout can have exogenous constraints (e.g., privacy) and is query dependent; that is, no one layout helps with all possible queries. In §5, we we will show that diPs offer significant additional speedup when used with the data layouts proposed by prior works and that diPs improve query performance in other layouts as well.

To the best of our knowledge, this paper is the first to offer data skipping gains across joins for complex queries before query execution while using only small per-table statistics. Prior work either only offers gains during query execution [21, 52, 55, 71, 76, 85] or uses more complex structures which have sizable maintenance overheads [5, 25, 48, 77, 82]. To achieve the above, we offer an efficient method to compute

Table 1: Data statistics maintained by several systems.

Scheme	Statistic	Granularity
ZoneMaps [13]	max and min value per column	*zone*
Spark [9, 17]		*file*
Exadata [67]	max, min and null present or null count per column	per table *region*
Vertica [61], ORC [2], Parquet [15]		stripe, row-group
Brighthouse [81]	histograms, char maps per col	*data pack*

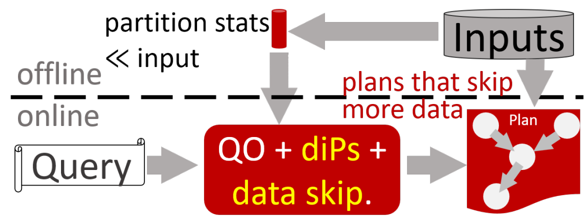


Figure 3: A workflow which shows changes in red; using partition statistics, our query optimizer computes data-induced predicates and outputs plans that read less input.

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 how to maintain statistics when datasets evolve. In more detail, the rest of this paper has these contributions.

- Using diPs for complex queries leads to novel challenges.
  - Consider TPC-H query#17 [19] in Figure 2(left) which has a nested sub-query on the `lineitem` table. Creating diPs for only the fragment considered in Figure 1 still reads the entire `lineitem` table for the nested group-by. To alleviate this, we use new QO transformation rules to move diPs; in this case, shown with dotted arrows, the diP is pulled above a join, moved sideways to a different join input and then pushed below a group-by thereby ensuring that a shared scan of `lineitem` will suffice.
  - When multiple joining tables have predicates, a second challenge arises. Consider the 3-way join in Figure 2(middle) where all tables have local predicates. The figure shows four diPs: one per table and per join condition. If applying these diPs eliminates partitions on some joining table, then the diPs that were previously constructed on that table are no longer up-to-date. Re-creating diPs whenever partition subsets change will increase data skipping but doing so naïvely can construct excessively many diPs which increases query optimization time. We present an optimal schedule for tree-like join graphs which converges to fixed point and hence achieves all possible data skipping while computing the fewest number of diPs. Star

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

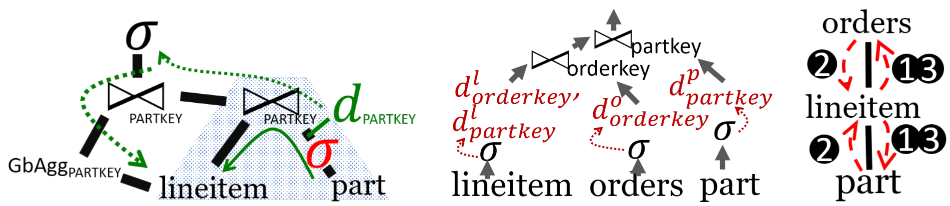


Figure 2: Illustrating the need to move diPs past other operations (left). On a 3-way join when all tables have predicates (middle), the optimal schedule only requires three (parallel) steps (right).

and snowflake schemas result in tree-like join graphs. We discuss how to derive diPs for general join graphs within a cost-based query optimizer in §3.

- We show how different data statistics can be used to compute diPs in §4 and discuss why *range-sets* represent a good trade-off between space usage and potential for data skipping.
- We discuss two methods to cope with dataset updates in §4.1. The first method *taints* 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.
- Fundamentally, data-induced predicates are beneficial only if the join column values in the partitions that satisfy a predicate contain only a small portion of all possible join column values. In §2.1, we discuss real-world use-cases that cause this property to hold and quantify their occurrence in production workloads.
- We report results from experiments on production clusters at Microsoft that have tens of thousands of servers. We also report results on SQL server. See Figure 3 for a high-level architecture diagram. Our results in §5 will show that using small statistics and a small increase in query optimization time, diPs offer sizable gains on three workloads (TPC-H [28], TPC-DS [26], JOB [12]) under a variety of conditions.

2 MOTIVATION

We begin with an example that illustrates how data-induced predicates (diPs) can enhance data skipping. Consider the query expression, $\sigma_{\texttt{year}}$ (`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 `year` is only available on the `date_dim` table, but the diPs are on the join column `date_sk` and can be pushed onto joining relations using column equivalence [55]. The diPs shown here are DNFs over ranges; if the equijoin condition has multiple columns, the diPs will be a conjunction of DNFs, one DNF per column. Further details on the class of predicates supported, extending to multiple joins and handling other operators, are in §3.2. Table 2b also shows that the diPs contain a small portion of the range of the join column `date_sk` (which is $[1000, 12000]$); thus, they can offer large data skipping gains on joining relations.

It is easy to see that diPs can be constructed using any data statistic that supports the following steps: (1) identify partitions that satisfy query predicates, (2) *merge* the data statistic of the join columns over the satisfying partitions, and (3) use the merged statistic to extract a new predicate that can identify partitions satisfying the predicate in joining relations. Many data statistics support these steps [33], and different stats can be used for different steps.

To illustrate the trade-offs in choice of data statistics, consider Figure 4a which shows equi-width histograms for the same columns and partitions as in Table 2a. A histogram with b buckets uses $b + 2$ doubles[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 non-overlapping ranges $\{[l_i, u_i]\}$ which contain all of the data values; such *range-sets* are a simple extension of zone-maps which are, trivially, range-sets of size 1. However, range-sets also record gaps that have no values. Figure 4b shows range-sets of size 2. It is

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

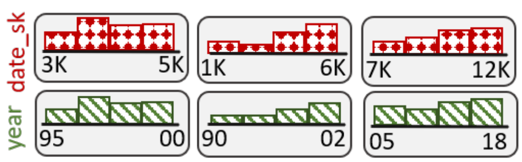
Table 2: Constructing diPs using partition statistics.

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

(a) Zone maps [13], i.e., the maximum and minimum values, for two columns in three hypothetical partitions of the `date_dim` table.

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

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



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

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

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

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

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 TPC-H query #17 [19] 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 [31], and tables were laid out in a typical manner[5]. Each partition is

#Partitions remaining in...

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

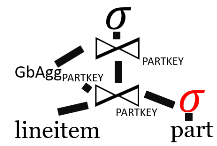


Figure 5: For TPC-H query 17 in Figure 2 (left), the table shows the partition reduction from using diPs. On the right, we show the plan generated using magic-set transformations which push group-by above the join. diPs complement magic-set transformations; we see here that magic-set tx cannot skip partitions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

~ 100MBs of data which is a typical quanta in distributed file systems [42] and is the default in our clusters [91]. Recall that the predicate columns are only available in the `part` table. The figure shows that only two partitions of `part` contain rows that satisfy the predicate and the corresponding diP eliminates many partitions in `lineitem`. We will show results in §5 for many different data layouts and data distributions. We discuss plan transformations needed to move the diP, as shown in Figure 2 (left), in §3.3. Overall, for the 100GB dataset, a 0.5MB statistic reduces the initial I/O for this query by 20×; the query can speed up by more or less depending on the work remaining after initial I/O.

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

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

- `C1` The predicate on a table is satisfied by rows belonging to a small subset of partitions of that table.
- `C2` The join column values in partitions that satisfy the predicate are a small subset of all possible join column values.
- `C3` In tables that receive diPs, the join column values are distributed such that diPs only pick a small subset of partitions.

We identify use-cases where these conditions hold based on our experiences in production clusters at Microsoft [47].

- Much of the data in production clusters is stored in the order in which it was ingested into the cluster [39, 43]. 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

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

[5] `lineitem` was clustered on `l_shipdate` and each cluster sorted on `l_orderkey`; `part` was sorted on its key; this layout is known to lead to good performance because it reduces re-partitioning for joins and allows date predicates to skip partitions [3, 22, 29].

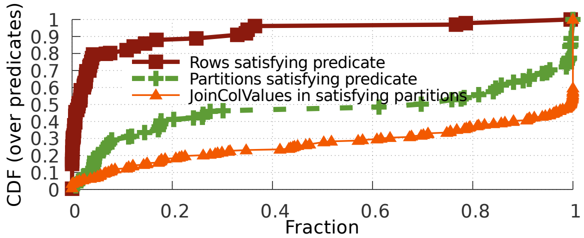


Figure 6: Quantifying how often the conditions that lead to large I/O skipping gains from using diPs hold in practice by using queries and datasets from production clusters at Microsoft.

few portions of the dataset. This helps with C1. When such datasets are joined on time or server-id, this phenomenon also helps with C2 and C3.

- A common physical design methodology for performant parallel plans is to hash partition a table by predicate columns and range partition or order by the join columns [3, 22, 29, 53] and vice-versa. Performance improves since the shuffles to re-partition for joins decrease [16, 49, 92] and predicates can skip data. Such data layouts help with all three conditions C1–C3 and, in our experiments, receive the largest I/O savings from diPs.
- Join columns are keys which monotonically increase as new data is inserted and hence are related to time. For example, both the title-id of movies and the name-id of actors in the IMDB dataset [11] roughly monotonically increase as each new title and new actor are added to the dataset. In such datasets, predicates on time as well as predicates that are implicitly related to time, such as co-stars, will select only a small range of join column values. This helps with C1 and C2.
- Practical datasets are skewed; often times the skew is heavy-tailed [34]. In skewed datasets, predicates and diPs that skip over the heavy hitters are highly selective; 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 cumulative distribution functions (CDFs) of the fraction of rows satisfying each predicate (red squares), the fraction of partitions containing these rows (green pluses) and the fraction of join column values contained in these partitions (orange triangles). We see that about 40% of the predicates pick less than 20% of partitions (C1)[6]; 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)[7].

Table 3: Notation used in this paper.

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

3 CONSTRUCTION AND USE OF DATA-INDUCED PREDICATES

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

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

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

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

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

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

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

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

[7] Read the value of the orange triangles line at $x = 0.5$ in Figure 6.

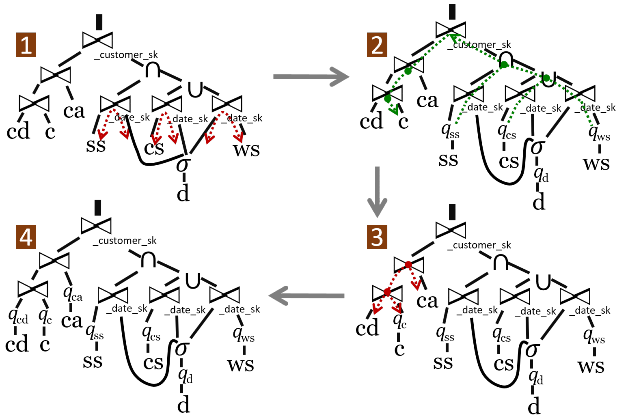


Figure 7: Illustrating the use of diPs in TPC-DS query #35. The table labels `ss`, `cs`, `ws` and `d` correspond to the tables `store_sales`, `catalog_sales`, `web_sales` and `date_dim`.

partitions can be eliminated). Acyclic join graphs can repeat this step up to $n-1$ times, i.e., construct up to $2m(n-1)$ diPs, and join graphs with cycles can take even longer (see 9.2 for an example). Abandoning this process before the partition subsets converge can leave data skipping gains untapped. On the other hand, generating too many diPs adds to query optimization time. To address this challenge, we construct diPs in a carefully chosen order so as to converge to the smallest partition subsets while building the minimum number of diPs (see §3.4).

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

3.1 Deriving diPs within a cost-based QO

Taken together, the previous paragraphs indicate two requirements to quickly identify efficient plans: (1) carefully schedule the order in which diPs are computed over a join graph and (2) use commutativity to move diPs past other operators in complex queries. We sketch our method to derive diPs within a cost-based QO here.

Let's consider some alternative designs. (1) Could the user or a query rewriting software that is separate from the QO insert optimal diPs into the query? This option is problematic because the user or the query rewriter will have to re-implement complex logic such as predicate simplification and push-down that is already available within the QO. Furthermore, moving diPs around other operators (see §3.3) requires plan transformation rules that are not implemented in today's QO; specifically rules that *pull up* diPs or *move them sideways* from one join input to another do not exist in typical QOs. As we saw with the case of the example in Figure 2(left), without such movement diPs may not achieve any data skipping. (2) Could the change to QO be limited to adding some new plan transformation rules? Doing so is appealing since the QO framework remains unchanged. Unfortunately, as we saw in the case of Figure 2(middle), diPs may have to be exchanged multiple times between the same pair of tables, and to keep costs manageable, diPs have to be constructed in a careful order over the join graphs; in today's cost-based optimizers, achieving such recursion and fine-grained query-wide ordering is challenging [55]. Thus, we use the hybrid design discussed next.

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

Example: Figure 7 illustrates this process for TPC-DS query #35; the SQL query is in [27]. As shown in the top left of the figure, labeled 1, diPs that are triggered by the predicate on `date_dim` are first exchanged in maximal SJ portions: `store_sales` ⋈ `date_dim`, `catalog_sales` ⋈ `date_dim` and `web_sales` ⋈ `date_dim`. The query joins these portions with another join expression after a few set operations. Hence, in 2, we build new diPs for the `customer_sk` column and pull those up through the set operations (union and intersection translate to logical or and and over diPs) and push down to the customer (c) table. To do so, we use the transformation rules in §3.3. In 3, if the incoming diP skips partitions on the customer table, another derivation ensues within the SJ expression on the left.[8] The final plan, shown in 4, effectively replaces each table with the partition subset that has to be read from that table.

3.2 Supported Queries

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

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

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

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

Projections: diPs commute trivially with any projection on columns that are not used in the diP. On columns that are used in a diP, only single-column invertible projections commute with that diP because only such projects can be inverted through zone-maps and other data statistics that we use to compute diPs[9].

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.

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

[9] Consider a single-column linear projection such as $\pi(x) = y = ax + b$ where a and b are constants, x is a column and y is a derived column; in this case the inverse is $\pi^{-1}(y) = x = \frac{y-b}{a}$. When the diP is on the same column as such a projection, we can pull up a changed diP. For example, if a diP is $x \in [0, 100]$ then $\pi(\text{diP}(x)) \equiv \text{diP}'(\pi(x))$ where diP' is $y \in [b, 100a + b]$. Generalizing from the above, similar transformations can be applied whenever projections are invertible, i.e., whenever a π^{-1} function exists that can be applied on the values in the range predicate that is the diP.

3.3 Commutativity of data-induced predicates with other operations

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

(1) diPs commute with any select.
(2) A diP commutes with any projection that does not affect the columns used in that diP. For projections that affect columns used in a diP, commutativity holds if and only if the projections are invertible functions on one column.
(3) diPs commute with a group-by if and only if the columns used in the diP are a subset of the group-by columns.
(4) diPs commute with set operations such as union, intersection, semi- and anti semi-joins, as shown below.
    - $d_1(\mathcal{R}_1) \cap d_2(\mathcal{R}_2) \equiv (d_1 \wedge d_2)(\mathcal{R}_1 \cap \mathcal{R}_2) \equiv (d_1 \wedge d_2)(\mathcal{R}_1) \cap (d_1 \wedge d_2)(\mathcal{R}_2)$
    - $d_1(\mathcal{R}_1) \cup d_2(\mathcal{R}_2) \equiv (d_1 \vee d_2)(d_1(\mathcal{R}_1) \cup d_2(\mathcal{R}_2))$
    - $d(\mathcal{R}_1) - \mathcal{R}_2 \equiv d(\mathcal{R}_1 - \mathcal{R}_2) \equiv d(\mathcal{R}_1) - d(\mathcal{R}_2)$
(5) diPs can move from one input of an equijoin to the other input if the columns used in the diP match the columns used in the equi-join condition. For outer-joins, a diP can move only if from the left side of a left outer join (and vice versa). No movement is possible with a full outer join.
    - $d_c(\mathcal{R}_1) \bowtie_{c=e} \mathcal{R}_2 \equiv d_c(\mathcal{R}_1 \bowtie_{c=e} \mathcal{R}_2) \equiv d_c(\mathcal{R}_1) \bowtie_{c=e} d_e(\mathcal{R}_2)$; note here that c and e can be sets of multiple columns, then $c = e$ implies set equality.

    Note a subtle case here: diPs whose columns do not match but are contained within the columns in the equi-join condition can also move.
(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 $\mathcal{R}_1$. Analogously, in one-sided outer joins, the diP on the join can move into the left side (right side) of a left outer join (right outer join) if the columns used in the diP are available on that join input.

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

3.4 Scheduling the deriving of predicates

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

Consider the example join graphs in Figure 8. The simple case of two tables on the left only requires a single exchange of diPs followed by an update to the partition subsets q; proof is in §9. The other two cases require more careful handling as we discuss next; the join graph in the middle is the popular star-join which leads to tree-like join graphs and on the right is a cyclic join graph.

Our algorithm to hasten convergence is shown in Pseudocode 9, Scheduler method at line#37. The case of acyclic join graphs is an important sub-case because it applies to queries with star or snowflake schema joins. Here, we construct a tree over the graph (Treeify in line#39 picks the root that has the smallest tree height and sets parent–child pointers; details are in §9.1). Then, we pass diPs up the tree (lines#7–#9) and afterwards pass diPs down the tree (lines#10–#12). To see why this converges, note that when line#10 begins, the partition subsets of the table at the root of the tree would have stabilized; Figure 8 (middle) illustrates this case with t_2 as root and shows that convergence requires at most two (parallel) epochs and six diPs. A proof that this algorithm is optimal, i.e., can skip all skippable partitions in all tables while constructing the fewest number of diPs, is in §9.

We convert a cyclic join graph into a tree over table subsets. The conversion retains completeness; that is, all partitions that can be skipped in the original join graph remain skippable by exchanging diPs only between adjacent table subsets on the tree. Furthermore, on the resulting tree, we apply the same simple schedule that we used above for tree-like join graphs with a few changes. For example, the join graph in Figure 8(right) becomes the following tree: $t_1 — \{t_2, t_3, t_5\} — \{t_3, t_4, t_5\} — \{t_4, t_5, t_6\} — t_7$.

Note that the join graph in Figure 8 (right) has two cycles one of which is not a clique. Figure 10 shows the process of converting this cyclic join graph into a tree using the junction tree algorithm. As shown in Figure 10b, we first triangulate the join graph; specifically, we add additional edges such that every cycle of four or more nodes has a chord (defined as an edge that connects two non-adjacent nodes in the cycle). Next, as

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

1 **Func:** DataPred $(q, \{c\})$ // Construct diP for columns $\{c\}$ over partitions x having q^x =1; see §4.
2 **Func:** Satisfy (d, x) // = 1 if partition x satisfies predicate d; 0 otherwise. See §4.
3 **Func:** Exchange(i, j) : //send diP from table i to table j
4 $d_{i \to j} \leftarrow$ DataPred$(q_i, \text{ColsOf}(p_{ij}, i))$
5 $\forall$ partition $x \in$ table $j, q_j^x \leftarrow q_j^x * \text{Satisfy}(d_{i \to j}, x)$
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:** ExchangeExt $(\mathcal{G}, u, v)$//send diPs from node u to v
14 **foreach** $t_1 \in$ RelationsOf(u), $t_2 \in$ RelationsOf(v) **do**
15   **if** IsConnected$(t_1, t_2, \mathcal{G})$ **then**
16         $d \leftarrow$ DataPred$(q_{t_1}, \text{ColsOf}(p_{t_1 t_2}, t_1))$;
17         $\forall$ partition $x \in$ table $t_2, q_{t_2}^x \leftarrow q_{t_2}^x * \text{Satisfy}(d, x)$
18 **Func:** ProcessNode (u) // exchange diPs within node
19 **for** $i \leftarrow 0$ **to** κ // *repeat up to* κ *times* **do**
20   change $\leftarrow$ false;
21   **foreach** *tables* $t_1, t_2 \in$ RelationsOf(u) **do**
22   **if** $(t_1 \neq t_2) \wedge$ IsConnected$(t_1, t_2, \mathcal{G})$ **then**
23             $d \leftarrow$ DataPred$(q_{t_1}, \text{ColsOf}(p_{t_1 t_2}, t_1))$;
24   **foreach** *partition* $x \in t_2 : q_{t_2}^x = 1$ **do**
25                 $q_{t_2}^x \leftarrow$ Satisfy(d, x)
26   change $\leftarrow$ change $\vee \left(q_{t_2}^x = 0\right)$
27   **if** $\neg$ change **then** break// no new pruning;
28 **Func:** TreeSchedulerExt $(\mathcal{V}, \mathcal{G}, \{q_i\})$// for cyclic join graphs.
29 **for** $h \leftarrow 0$ **to** height$(\mathcal{V}) - 1$ // *bottom-up traversal* **do**
30   **foreach** $u \in \mathcal{V}$: height$(u) = h$ **do**
31   **if** IsNotSingleRelation(u) **then** ProcessNode(u);
32   ExchangeExt$(\mathcal{G}, u$, Parent$(u, \mathcal{V}))$;
33 **for** $h \leftarrow$ *height*$(\mathcal{V})$ **to** 0 // *top-down traversal* **do**
34   **foreach** $u \in \mathcal{V}$: height$(t) = h$ **do**
35   **if** IsNotSingleRelation(u) **then** ProcessNode(u);
36         $\forall$ child v of u in $\mathcal{V}$, ExchangeExt$(\mathcal{G}, u, v)$;
37 **Func:** Scheduler$(\mathcal{G}, \{q_i\})$:
38 **if** IsTree$(\mathcal{G})$ **then**
39   **return** TreeScheduler (Treeify$(\mathcal{G})$, $\{q_i\}$);
40 **else**
41     $\mathcal{V} \leftarrow$ MaxWtSpanTree(CliqueGraph(Triangulate$(\mathcal{G})$))
42   **return** TreeSchedulerExt(Treeify$(\mathcal{V})$, $\mathcal{G}$, $\{q_i\}$);

Figure 9: Pseudocode to compute a fast schedule.

shown in Figure 10c, we construct a *clique-graph* consisting of nodes that are maximal cliques in the triangulated join graph and edges between nodes that contain the same relations or relations that are connected in the original join graph. The

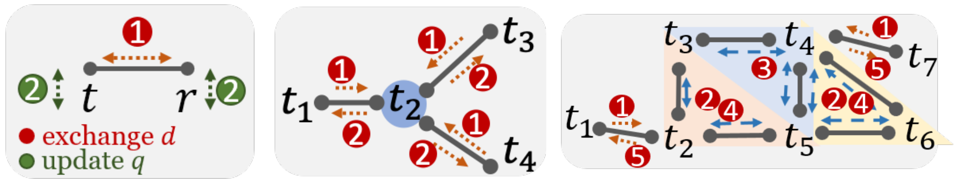
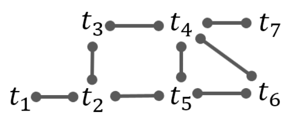
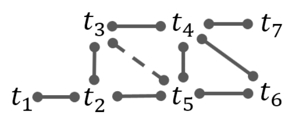


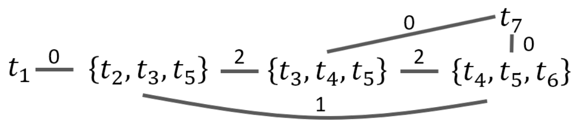
Figure 8: **Schedules of exchanging diPs for different join graphs; numbers-in-circles denote the epoch; multiple diPs are exchanged in parallel in each epoch. Details are in §3.4.**



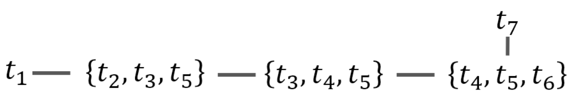
(a) **A join graph $\mathcal{G}$.**



(b) Triangulate($\mathcal{G}$).



(c) **Result of the** CliqueGraph **method over the triangulated graph above; edge weight is the number of common tables between nodes incident on the edge.**



(d) **Illustrating the** MaxWtSpanTree **method for the above clique graph; the schedule is in Figure 8 (right).**

Figure 10: **Illustrating the steps in lines#41,#42 for a join graph with seven relations and two cycles.**

weight of an edge is the number of relations that are common between the incident nodes. Lastly, as shown in Figure 10d, we compute the maximum weighted spanning tree over the weighted clique graph. The net effect is to translate the cyclic join graph into a tree of nodes corresponding to subsets of connected relations in the join graph. On the resulting tree we mimic the strategy used for tree-like join graphs with two key differences. Specifically, at line#42, Treeify picks a root with the lowest height as before. Then, diPs are exchanged from children to parents (lines#29–#32) and from parents to children (lines#33–#36). The key differences between the TreeSchedulerExt and the TreeScheduler methods are: (1) as the ProcessNode method shows, diPs are exchanged until convergence or at most κ times between relations that are contained in a node and (2) we compute multiple diPs when exchanging information between nodes (see ExchangeExt) whereas the Exchange method constructs at most one diP. Figure 8 (right) illustrates the resulting schedule; the root shown in blue is the node containing $\{t_3, t_4, t_5\}$; epochs #2, #3 and #4 invoke ProcessNode on the triangle subsets of tables which have the same color whereas epochs #1 and #5 exchange at most one diP on the edges shown.

Properties of Algorithm 9: For tree-like join graphs, the method shown is optimal (proof in §9). For a tree-like join graph $\mathcal{G}$ with n tables, this method computes at most $2(n-1)$ diPs (because a tree has $n-1$ edges) and requires 2height($\mathcal{G}$) (parallel) epochs where tree height can vary from $\lceil \frac{n}{2} \rceil$ to $\lceil \log n \rceil$. For cyclic join graphs the method shown here is approximate; that is, it will not eliminate all partitions that can be skipped. We show by counter-example in §9.2 that the optimal schedule for a cyclic join graph can require a very large number of diPs; the sub-optimality arises from limiting how often diPs are exchanged between relations within a node (in the ProcessNode method). In §5.4, we empirically demonstrate that our method for cyclic join graphs is a good trade-off between achieving large data skipping and computing many diPs.

4 USING STATISTICS TO BUILD DIPS

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

Zone-maps [13] consist of the minimum and maximum value per column per partition and are maintained by several systems today (see Table 1). Each predicate clause listed in §3.2 translates to a logical operation over the zone-maps of the columns involved in the predicate. Conjunctions, disjunctions and negations translate to an intersection, an union or set difference respectively over the partition subsets that match each clause. For string-valued columns, zone-maps are typically built over hash values of the strings and so equality check is also a logical equality, but regular expressions are not supported.

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

The diP constructed using zone-maps, as we saw in the example in Table 2b, is a union of the zone-maps of the partitions satisfying the predicate; hence, the diP is a disjunction over

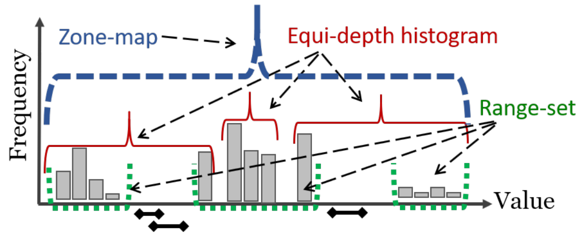


Figure 11: Illustrating the difference between range-sets, zone-maps and equi-depth histogram; both histograms and range-sets have three buckets. The predicates shown in black dumbels below the axes will be false positives for all stats except the range-set.

non-overlapping ranges. On the table that receives a diP, a partition will satisfy the diP only if there is an overlap between the diP and the zone-map of that partition. Note that there can be false positives in this check as well because no actual data row may have a value within the range that overlaps between the diP and the partition's zone map. It is straightforward to implement these checks efficiently, and our results will show that zone-maps offer sizable I/O savings (Figures 12, 15).

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 [50] 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 [32, 50, 54], are expensive to maintain [54], and the frequency information in histograms, while useful for other purposes, is a waste of space here because predicate satisfaction and diP construction only check for the existence of values.

Bloom filters record set membership [41]. However, we found them to be less useful here because the partition sizes used in practical distributed storage systems (e.g., ~ 100MBs of data [42, 91]) result in millions of distinct values per column in each partition, especially when join columns are keys. To record large sets, bloom filters require large space or they will have a high false positive rate; e.g., a 1KB bloom filter that records a million distinct values will have 99.62% false positives [41] leading to almost no data skipping.

Alternatives such as the count-min [51] and AMS [38] sketches behave similarly to a bloom filter for the purpose at hand. Their space requirement is larger, and they are better at capturing the frequency of values (in addition to set membership). However, as we noted in the case of histograms, frequency information is not helpful to construct diPs.

Range-set: To reduce false-positives while keeping the stat size small, we propose storing a set of non-overlapping ranges over the column value, $\{[l_i, u_i]\}$. Note that a zone-map is a range-set of size 1; using more ranges is hence a simple generalization. The boundaries of the ranges are chosen to reduce false positives by minimizing the total width (i.e., $\sum_i u_i - l_i$) while covering all of the column values. To see why range-sets help, consider the range-set shown in green dots in Figure 11; compared to zone-maps, range-sets have fewer false positives because they record empty spaces or gaps. Equi-depth histograms, as the figure shows, will choose narrow buckets near more frequent values and wider buckets elsewhere which can lead to more false positives. Constructing a range-set over r values takes $O(r \log r)$ time[10]. Reflecting on how zone-maps were used for the three operations in Equations 1– 3, i.e., applying predicates, constructing diPs and applying diPs on joining tables, note that a similar logic extends to the case of a range-set. SIMD-aware implementations can improve efficiency by operating on multiple ranges at once. A range-set having n ranges uses $2n$ doubles. Merging two unsorted range-sets as well as checking for overlap between them uses $O(n \log n)$ time where n is the size of larger rangeset; proof is in §10[11]. Our results will show that small numbers of ranges (e.g., 4 or 20) lead to substantial improvements over zone-maps (Figure 18).

4.1 Coping with data updates

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

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

More specifically, the operations over statistics (Equations 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.

$$\forall \text{ table } i, \text{ partition } x, \qquad q_i^x \leftarrow t_i^x \vee \mathsf{Satisfy}(p_i, x), \tag{4}$$
$$\forall \text{ tables } i, j, \text{ if } \forall x, t_i^x = 0, \qquad d_{i\rightarrow j} \leftarrow \mathsf{DataPred}(q_i, p_{ij}), \tag{5}$$
$$\forall \text{ table } j, \text{ partition } x, \qquad q_j^x \leftarrow t_j^x \vee q_j^x \textstyle\prod_{i\neq j} \mathsf{Satisfy}(d_{i\rightarrow j}, x). \tag{6}$$

Taint bits can be maintained at transactional speeds and can be extremely effective in some cases, e.g., when updates are mostly to tables that do not generate data-reductive diPs.

[10] 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 §10 for proof of optimality.
[11] The sorting cost can be amortized; e.g., by sorting after any update, so that merge and check are in practice ~ $O(n)$.

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

Beginning range-set: $\{[3,5],[10,20],[23,27]\}$, $n_r = 3$

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

One such scenario is queries over updateable *fact* tables that join with many unchanging *dimension* tables; predicates on dimension tables can generate diPs that flow unimpeded by taint on to the fact tables. Going beyond one taint bit per partition, maintaining taint bits at a finer granularity (e.g., per partition and per column) can improve performance with a small increase in update cost. See results in §5.3. 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; otherwise, either grow an existing range to contain the new value or merge two consecutive ranges and add the new value as a new range all by itself. Since these options increase the total width of the range-set, the process greedily chooses whichever option has the smallest increase in total width. Table 4 shows examples of greedily growing a *range-set*. Our results will show that such an update is fast (Table 7), and the reduction in I/O savings— because the range-sets after several such updates can have more false positives than range-sets that are re-constructed for just the new column values—is small (Figure 16a).

We also have hardness results regarding the non-existence of an optimal data statistic for diPs in §10; 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 §15).

5 EVALUATION

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

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

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

5.1 Methodology

Queries: We report results on TPC-H [28], TPC-DS [26] and the join order benchmark (JOB) [63]. 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[12]. 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.

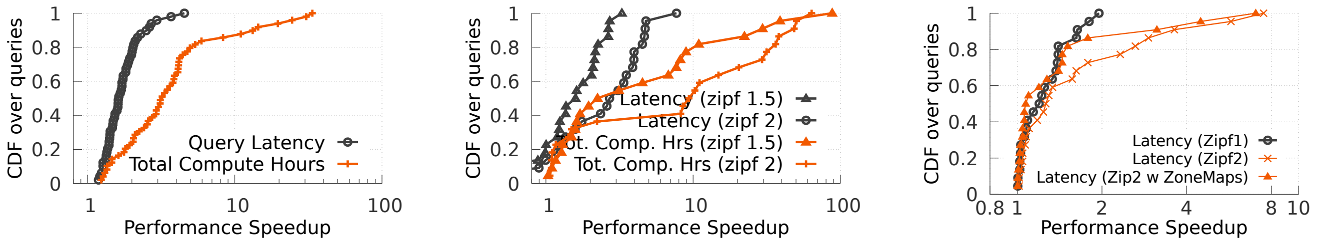
Datasets: For TPC-H and TPC-DS we use 100GB and 1TB datasets respectively. The default datagen for TPC-H, unlike that of TPC-DS, creates uniformly distributed datasets which is not representative of practical datasets; therefore, we also use a modified datagen [31] to create datasets with different amounts of skew (e.g., with zipf factors of $1, 1.5, 2$). For JOB, we use the IMDB dataset from May 2013 [63].

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[13]. 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, [47, 91] and roughly 1M rows per columnstore segment in SQL Server [8].

Systems: We have built prototypes on top of two production platforms: SCOPE clusters which serve as the primary platform for batch analytics at Microsoft and comprise tens of thousands of servers [47, 91] and SQL Server 2016. Both

[12] $1 \ldots 40, 90 \ldots 99$ from TPC-DS and $([1-9]|10)^*$ from JOB

[13] 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 §13 .



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

systems use cost-based query optimizers [55]. A SCOPE job is a collection of tasks orchestrated by a job manager; tasks read and write to a file system, and each task internally executes a sub-graph of relational operators which pass data through memory. The servers are state-of-the-art Intel Xeons with 192GB RAM, multiple disks and multiple 10Gbps network interface cards. Our SQL server experiments ran on a similar server. After each query executes in SQL server, we flush various system buffer pools to accurately measure the effects of I/O savings. SCOPE clusters use a partitioned row store; for SQL server, we use both columnstores and rowstores. SCOPE and SQL server implement several advanced optimizations such as semijoins [21], predicate pushdown to eliminate partitions [7] and magic-set rewrites [52].

Comparisons: In addition to the above production baselines, we compare against several alternatives. By DenormView, we refer to a technique that avoids joins by *denormalization*, i.e., materializes a join view over multiple tables. The view is stored in column store format in SQL server. Since the view is a single relation, queries can skip partitions without worrying about joins. By JoinIndexes, we refer to a technique that maintains clustered rowstore indexes on the join columns of each relation; for tables that join on more than one column, we build an index on the most frequently used 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 [82]. We apply FineBlock on the above denormalized view.

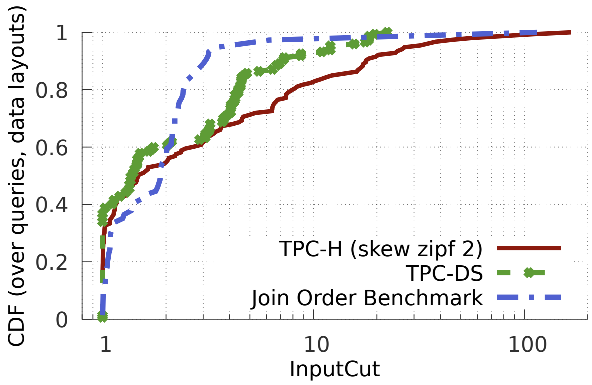
We also compare with the following variants of our scheme: No Transforms does not use commutativity to move diPs; Naive Schedule constructs as many diPs as our schedule but picks at random which diP to construct at each step. Preds uses the same statistics but only for predicate pushdown, i.e., it does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

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

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

5.2 Benefits from using diPs

Figure 12 shows the performance speedup from using diPs on different workloads in SCOPE clusters and SQL server. Results are on the tuned layout which is popular because it avoids re-partitioning for joins and enhances data skipping [22, 29, 53]. 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 12c use range-sets. Figure 12a 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 [49]. We see that almost all queries improve. SCOPE clusters are shared by over hundreds of concurrent jobs, and so query latency is subject to performance interference; the CDFs use the median value over at least five trials, but some TPC-H queries in Figure 12b still have a small regression in latency. Figures 12b and 12c 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



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 13: 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).

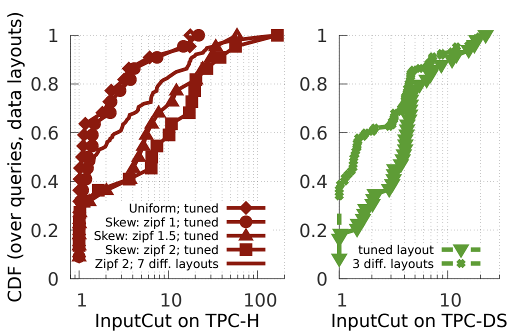


Figure 14: How input skew and data layouts affect the usefulness of diPs; see §5.2.

show results with different amounts of skew generated using [31]. We see that diPs produce larger speed-ups as skew increases mainly because predicates and diPs become more selective at larger skew. Figure 12c shows sizable latency improvements when using zone-maps. We have confirmed that the query plans in the production systems, SCOPE clusters and SQL server, reflect the effects of predicate pushdown and bitmap filters for semijoins [7, 21, 52]; these figures show that diPs offer sizable gains for a sizable fraction of benchmark queries on top of such optimizations.

Figure 13 considers many different layouts, and Figure 14 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

Table 5: The additional latency to derive diPs in seconds compared to the baseline QO latency; see §5.3.

%ile / Latency(s)	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 6: Additional results for experiments in Figure 12, Figure 13 and Figure 14. The table shows data from our SCOPE cluster.

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

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

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

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 14 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 15 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, an increase to the query optimization duration (to determine which partitions can be skipped), and maintaining statistics when data changes. In big-data clusters, queries are read-only

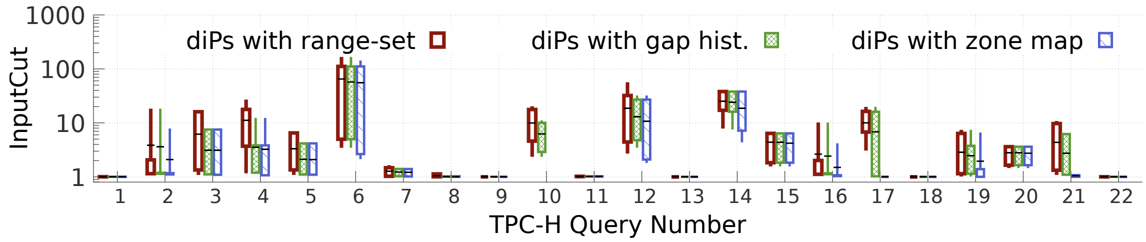
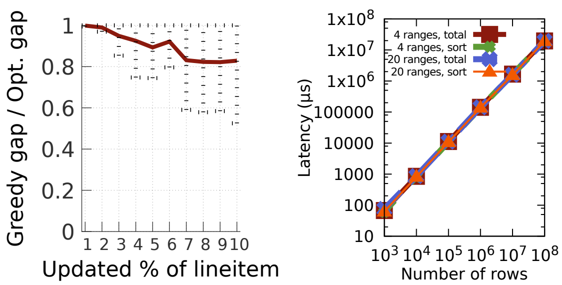


Figure 15: **The figure shows the InputCut values when using different stats to construct** diPs **(for TPC-H with zipf 2 skew). Candlesticks show variation across seven different data layouts including the** tuned **layout; the rectangle goes from 25th to 75th percentile, the whiskers go from min to max and the thin black dash indicates the average value. Zone-maps do quite well and range-sets are a sizable improvement.**



(a) Greedy updates. **(b) Construction latency**

Figure 16: **(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.**

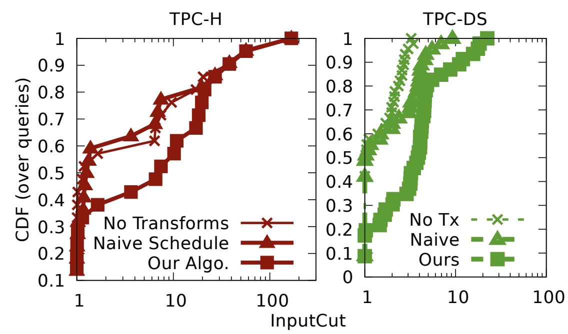


Figure 17: **How InputCut varies when different methods are used to derive** diPs**; we compare with** No Transforms **which does not use any transformation rules and** Naive schedule **which constructs the same number of** diPs **in a naive manner. (Results are for TPC-H skewed with zipf 2 and TPC-DS in the** tuned **layout.)**

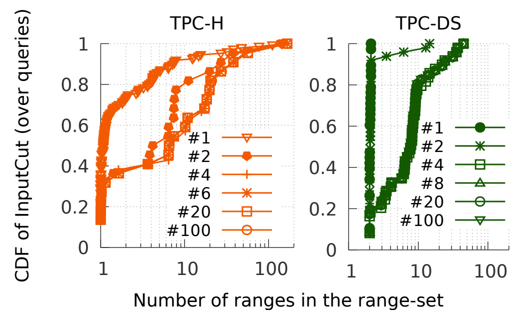


Figure 18: **How InputCut varies with the numbers of ranges used in the range-set statistic. (Results are for TPC-H skewed with zipf 2 and TPC-DS in the** tuned **layout; other cases behave similarly.)**

and datasets are bulk appended; so building and maintaining statistics is less impactful relative to the 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 tables which takes a long time because such diPs have many clauses and are evaluated on many partitions. We note that our derivation of diPs is a prototype, parts of which are in c# for ease-of-debugging, and that evaluating diPs is embarrassingly parallel (e.g., apply diP to stat of each partition); we have not yet implemented optimizations and believe that the extra QO time can be substantially smaller. Regarding storage overhead, Table 6 shows the size of range-sets which can be thought of as 20 "rows" per partition (a partition is 100MB of data in SCOPE clusters and 1M rows in a columnstore segment in SQL server [8]) and so the space overhead for range-sets is ~ 0.002%. Zone-maps use 10× less space because they only record the max and min value per column, i.e., 2 "rows" per partition. Although TPC-DS and JOB have more tables and more columns, their ratio of stat size to input size is similar.

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

Greedily maintaining range-sets: Recall from §4 that our second proposal to cope with data updates is to greedily grow the range-set statistic to cover the new values. Table 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 16a shows that the greedy update procedure leads to a reasonably high quality range-set statistic; that is, the total gap value (i.e., $\sum_i (u_i - l_i)$ for a range-set $\{[l_i, u_i]\}$) obtained after many greedy updates is close to the total gap value of an optimal range-set constructed over the updated dataset. The figure shows that the greedy updates lead to a range-set with an average gap value ≥ 80% of optimal when up to 10% of rows in the `lineitem` table are updated.

Range set construction time: Figure 16b shows the latency to construct range-sets. Computing larger range-sets (e.g., 20 ranges vs. 4) has only a small impact on latency, and almost all of the latency is due to sorting the input once (the 'total' lines are indistinguishable from the 'sort' lines). These results use std::sort from Microsoft Visual C++. Note that range-sets can be constructed during data ingestion in parallel per partition and column; construction can also piggy-back on the first query to scan the input.

5.4 Understanding why diPs help

Comparing different methods to construct diPs: Figure 17 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 18 shows that a small number of ranges achieve nearly the same amount of data skipping as much larger range-sets. Each step in diP creation, as noted in §4, adds false positives, and there is a limit to gains based on the joint distribution of join and predicate columns. We believe that achieving more I/O skipping beyond that obtained by using just a few ranges may require much larger statistics and/or more complex techniques.

Drilling down on why diPs help: To understand the result in Figure 15 further, we assess how often the conditions C1–C3 noted in §2.1 for when diPs yield large gains, 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**}.

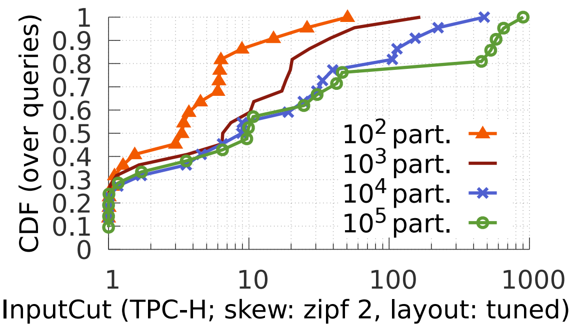


Figure 19: Impact of changing the number of partitions for the TPC-H dataset (scale factor = 100).

- 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 9 offers additional detail with more detailed conditions than those mentioned in §2.1.

Effect of changing #partitions: Figure 19 shows the results for TPC-H queries when varying the number of partitions. Recall that all the other results in the paper used 10^3 partitions for TPC-H (see Table 6); here, we show results for fewer and many more partitions.

The figure shows that InputCut, which is the fraction of partitions that can be skipped using diPs, improves as the number of partitions increase but the marginal improvement decreases. The reason is that more partitions allow a finely granular view of the relations which can lead to more partitions being skipped; however, if a certain number of partitions suffices to skip all of the skippable rows for a query then dividing the data into more partitions will not offer more data skipping.

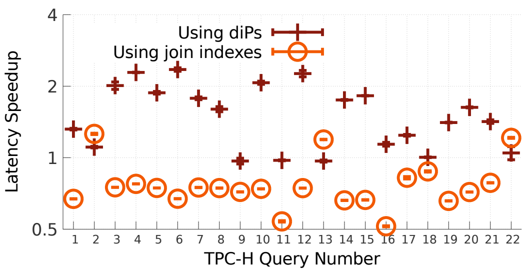


Figure 20: Comparing diPs with join indices on SQL server.

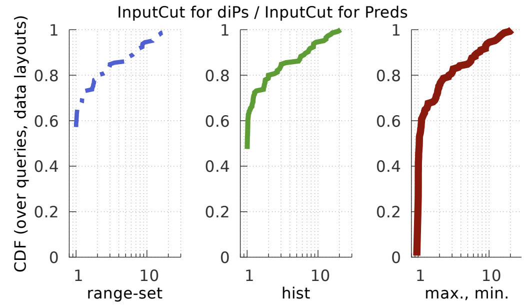


Figure 21: Comparing diPs versus using the same stats to only skip partitions on individual tables.

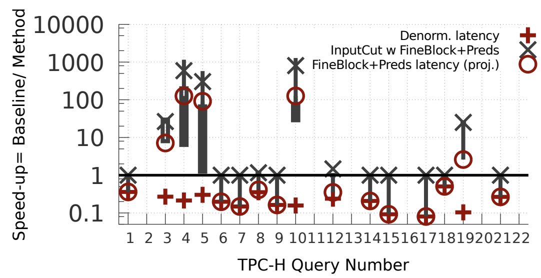


Figure 22: InputCut on TPC-H queries for FineBlock; box plots show results for different predicates.

While using more partitions generally improves data skipping, some of the associated costs increase as noted below:

- A significant concern arises from I/O access: increasing the number of partitions naïvely could lead to small partition sizes (at extremum, each partition has just 1 row) and reading small amounts of data is inefficient in most batch storage devices. For example, on a disk that uses 4KB blocks, the maximal disk throughput may not be realized until about 400KBs of data is read sequentially. Thus, very small partitions may lead to worse query execution time even though they facilitate greater data skipping. Recall that we use a 100GB dataset for TPC-H and so even with 10^5 partitions, each partition is roughly 1MB. Thus, more gains are possible than is indicated by our experimental results in §5.
- The query optimization time to construct and apply diPs can also increase because all active partitions are inspected in these operations. In practice, the increase is not significant because diP construction and application are easily parallelizable (e.g., an incoming diP can be evaluated on the statistics of each partition in parallel).
- Lastly, the complexity of computing statistics does not change (recall that this is linear in the number of rows) but the space requirements to store statistics increases (this is linear in the number of the partitions).

5.5 Comparing with alternatives

Join Indexes: Figure 20 compares using diPs with the JoinIndexes scheme described in §5.1. Results are on SQL server for TPC-H skewed with zipf factor 1 and a scale factor of 100. We built clustered rowstore indexes [6] on the key columns of the dimension tables, and on the fact tables, we built clustered indexes on their most frequently used join columns (i.e., `l_orderkey`, `o_orderkey`, `ps_partkey`). Indexes are not supported on columnstores; so we use rowstores for just this experiment. The figure shows that using join indexes leads to worse query latency than not using the indexes in 19/22 queries; we believe that this is because: (1) the predicate selectivity in several TPC-H queries is not small enough to benefit from an index seek and so most plans use a clustered index scan, and (2) clustered index scans are slower than table scans. diPs are complementary because they reduce I/O before query execution.

diPs **vs. predicate pushdown:** Figure 21 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 §14). In columnar (rowstore) format, this view occupies 2.7× (6.2×) more storage space than all of the tables combined. Queries over the view are 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 all tables are in a columnar layout. Figure 22 (with + symbol) shows the speed-up in query latency when using this view in SQL server (results are for 100G dataset with zipf skew 1). We see that all of the queries slow down (all + symbols are below 1). Materialized views speed-up queries, in general, only when the views have selections or aggregations that reduce the size of the view [36]. Unfortunately, this does not happen in the case of the view that subsumes all 16/22 TPC-H queries (see [18]).

diPs **vs. clustering rows by predicates:** A recent research proposal [82] clusters rows in the above view to maximize data skipping. Training over a set of predicates, [82] learns a clustering scheme that intends to skip data for unseen predicates. Figure 22 shows with × symbols the average InputCut obtained as a result of such clustering; the candlesticks around the × symbol show the min, 25th percentile, 75th percentile and max InputCut for different query predicates. We see that most queries receive no InputCut (× marks are at 1) due primarily to two reasons: (1) the chosen clustering scheme does not generalize across queries; that is, while some queries receive gains the chosen clustering of rows does not help all queries, and (2) the chosen clustering scheme does not generalize to unseen predicates as can be seen from the large span of the candlesticks. Figure 22 also shows with circle symbols the average query latency when using this clustering. Only 5/22 queries improve (fastest query is ~ 100× faster) and 11/22 queries regress (slowest is 10× slower). Hence, the practical value of such schemes is unclear.

We also note the rather large overheads to create and maintain indexes and views [35, 72] and to learn clusterings [82]. Also, these schemes require foreknowledge of queries and offer gains only for queries that are similar [36, 37, 82]. In contrast, diPs only use small and easily maintainable data statistics, require no apriori knowledge of queries and offer sizable gains for ad-hoc and complex queries.

6 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 [75]. A diP can also be sent to a remote store [23] 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 [21].

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 [62].

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 §9 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: When datasets change, query execution plans that were cached [24] when the same or a similar query executed earlier are no longer up-to-date. Nevertheless, some systems reuse the cached query plan as long as the number of changes or the number of affected rows are small. The reasoning here is that a small number of changes may not affect the quality of the plan, and avoiding re-optimization may be a worthwhile trade-off. More care is required, however, when re-using cached query plans that contain diPs. If the changes to data affect the diPs that were used to build the plan, then using the cached plans may lead to an incorrect query result. Here, we mention a simple method that SQL server uses to keep plan caches up-to-date with the data statistics used to build such plans, and present a simple extension to consider the case of plans containing diPs. For every query plan stored in the plan cache, SQL server maintains a list of *interesting statistics* that were used to generate that plan. Associated with each interesting statistic, SQL server maintains a counter that describes how often the table corresponding to that statistic was updated. A threshold function is used on these modification counters to determine whether the cached plan can be reused. To extend this method to the case of query plans that contain diPs, we propose to add the data statistic used to compute the diPs as another interesting statistic; the counter is updated if the corresponding table receives a taint on any partition or if growing the statistic changes its value, and we choose a threshold value such that plans are reused if and only if the underlying data statistics do not change or if the tables are untainted as the case may be. Caching plans containing diPs can help in certain common use-cases such as read-only queries which are predominant in big-data clusters and append-only semantics for datasets which are common in columnstore deployments. In other cases, the above small extension to SQL server's current operation suffices to ensure reuse of cached query plans.

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, are built on-the-fly in the QO, can skip partitions of multiple joining relations, support different join types and work with complex queries; the resulting plans only read subsets of the input relations and have no execution-time overhead.

Some research works discover data properties such as functional dependencies and column correlations and use them to improve query plans [10, 44, 58, 60]. Inferring such data properties is a sizable cost (e.g., [58] uses student t-test between every pair of columns). It is unclear if these properties can be maintained when data evolves. More importantly, imprecise data properties are less useful for QO (e.g., a *soft* functional dependency does not preserve set multiplicity and hence cannot guarantee correctness of certain plan transformations over group-bys and joins). A SQL server option [10] uses the fact that the `l_shipdate` attribute of `lineitem` is between 0 to 90 days larger than `o_orderdate` from `orders` [28] to convert predicates on `l_shipdate` to predicates on `o_orderdate` and vice versa. Others discover similar constraints more broadly [44, 60]. In contrast, diPs exploit relationships that may only hold conditionally given a query and a data-layout. Specifically, even if the predicate columns and join columns are independent, diPs can offer gains if the subset of partitions that satisfy a predicate contain a small subset of values of the join columns. As we saw in §2, such situations arise when datasets are clustered on time or partitioned on join columns [53].

Prior work moves predicates around using column equivalence and magic-set style reasoning [52, 64, 66, 76, 85, 86]. SCOPE clusters and SQL server implement such optimizations, and as we saw in §5, diPs offer gains over these baselines. Column equivalence does not help when predicate columns do not exist in joining relations. Magic set transformations help only 2/22 queries in TPC-H queries and only when predicates are selective [76]. By inferring new predicates that are induced by data statistics, diPs have a wider appeal.

Auxiliary data structures such as views [30], join indices [25], join bitmap indexes [5], succinct tries [90], column sketches [56], and partial histograms [87] can also help speed-up queries. Join zone maps [14] on a fact table can be constructed to include predicate columns from dimension tables; doing so effectively creates zone-maps on a larger denormalized view. Constructing and maintaining these data structures has overhead, and as we saw in §5, a particular view or join index does not subsume all queries. Hence, many different structures are needed to cover a large subset of queries which further increases overhead. Queries with foreign-key — foreign-key joins (e.g., `store_sales` and `store_returns` in TPC-DS join in six different ways) can require maintaining many different structures. diPs can be thought off as a complementary approach that helps with or without auxiliary structures.

While data-induced predicates are similar to the implied integrity constraints used by [68], there are some key differences and additional contributions. (1) [68] only exchanges constraints between a pair of relations; we offer a method which exchanges diPs between multiple relations, handles cyclic joins and supports queries having group-by's, union's and other operations. (2) [68] uses zone maps and two bucket histograms; we offer a new statistic (range-set) that performs better. (3) [68] shows no query performance improvements; we show speed-ups in both a big-data cluster and a DBMS. (4) [68] offers no results in the presence of data updates; we design and evaluate two maintenance techniques that can be built into transactional systems.

While a query executes, sideways information passing (SIP) from one sub-expression to a joining sub-expression can prune the data-in-flight and speed up joins [40, 59, 66, 71, 76, 79]. Several systems, including SQL server, implement SIP and we saw in §5 that diPs offer additional speed-up. This is because SIP only applies during query execution whereas diPs reduce the I/O to be read from the store. SIP can reduce the cost of a join, but constructing the necessary info at runtime (e.g., a bloom filter over the join column values from one input) adds runtime overhead, needs large structures to avoid false positives and introduces a barrier that prevents simultaneous parallel computation of the joining relations. Also, unlike diPs, SIP cannot exchange information in both directions between joining relations nor does it create new predicates that can be pushed below group-bys, unions and other operations.

A large area of related work improves data skipping using workload aware adaptations to data partitioning or indexing [45, 46, 57, 65, 69, 74, 78, 82, 83, 89]; they co-locate data that is accessed together or build correlated indices. Some use denormalization to avoid joins [82, 89]. 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.

REFERENCES

[1] 2017 big-data and analytics forecast. https://bit.ly/2TtKyjB.
[2] Apache orc spec. v1. https://bit.ly/2J5BIkh.
[3] Apache spark join guidelines and performance tuning. https://bit.ly/2Jd87We.
[4] Band join. https://bit.ly/2kixJJn.
[5] Bitmap join indexes in oracle. https://bit.ly/2TLBBTF.
[6] Clustered and nonclustered indexes described. https://bit.ly/2Drdb90.
[7] Columnstore index performance: Rowgroup elimination. https://bit.ly/2VFpljV.
[8] Columnstore indexes described. https://bit.ly/2F7LZuI.
[9] Data skipping index in spark. https://bit.ly/2qONacb.
[10] Date correlation optimzation in sql server 2005 & 2008. https://bit.ly/2VodSVN.
[11] Imdb datasets. https://imdb.to/2S3BzSF.
[12] Join order benchmark. https://bit.ly/2tTRyIb.
[13] Oracle database guide: Using zone maps. https://bit.ly/2qMeO9E.
[14] Oracle: Using zone maps. https://bit.ly/2vsUWKK.
[15] Parquet thrift format. https://bit.ly/2vm6D5U.
[16] Presto: Repartitioned and replicated joins. https://bit.ly/2JauYll.
[17] Processing petabytes of data in seconds with databricks delta. https://bit.ly/2Pryf2E.
[18] Pushing data-induced predicates through joins in bigdata clusters; extended version. https://bit.ly/2WhTwP1.
[19] Query 17 in tpc-h, see page #57. https://bit.ly/2kJRV72.
[20] Query 1a in job. https://bit.ly/2Fomtmx.
[21] Query execution bitmap filters. https://bit.ly/2NJzzgF.
[22] Redshift: Choosing the best sort key. https://amzn.to/2AmYbXh.
[23] S3 sequential scan. https://amzn.to/2PHd38g.
[24] Sql server, plan cache object. https://bit.ly/2E4cBgH.
[25] Teradata: Join index. https://bit.ly/2FbalDT.
[26] TPC-DS Benchmark. http://www.tpc.org/tpcds/.
[27] Tpc-ds query #35. https://bit.ly/2UorIk6.
[28] TPC-H Benchmark. http://www.tpc.org/tpch.
[29] Vertica: Choosing sort order: Best practices. https://bit.ly/2yrvPtG.
[30] Views in sql server. https://bit.ly/2CnbmIo.
[31] Program for tpc-h data generation with skew. https://bit.ly/2wvdNVo, 2016.
[32] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *SIGMOD*, 1999.
[33] P. K. Agarwal et al. Mergeable summaries. *TODS*, 2013.
[34] S. Agarwal et al. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.
[35] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient view maintenance at data warehouses. In *SIGMOD*, 1997.
[36] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. *VLDB*, 2000.
[37] S. Agrawal et al. Database tuning advisor for microsoft sql server 2005. *VLDB*, 2004.
[38] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *STOC*, 1999.
[39] M. Armbrust et al. Spark sql: Relational data processing in spark. In *SIGMOD*, 2015.
[40] F. Bancilhon et al. Magic sets and other strange ways to implement logic programs. In *SIGMOD*, 1985.
[41] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 1970.
[42] D. Borthakur et al. Hdfs architecture guide. *Hadoop Apache Project*, 2008.
[43] D. Borthakur et al. Apache hadoop goes realtime at facebook. In *SIGMOD*, 2011.
[44] P. G. Brown and P. J. Haas. Bhunt: Automatic discovery of fuzzy algebraic constraints in relational data. In *VLDB*, 2003.
[45] M. Brucato, A. Abouzied, and A. Meliou. A scalable execution engine for package queries. *SIGMOD Rec.*, 2017.
[46] L. Cao and E. A. Rundensteiner. High performance stream query processing with correlation-aware partitioning. *VLDB*, 2013.
[47] R. Chaiken et al. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *VLDB*, 2008.
[48] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 2012.
[49] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD*, 2015.
[50] 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.
[51] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 2005.
[52] M. Elhemali, C. A. Galindo-Legaria, T. Grabs, and M. M. Joshi. Execution strategies for sql subqueries. In *SIGMOD*, 2007.
[53] M. Y. Eltabakh et al. Cohadoop: Flexible data placement and its exploitation in hadoop. In *VLDB*, 2011.
[54] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. In *VLDB*, 1997.
[55] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 1995.
[56] B. Hentschel, M. S. Kester, and S. Idreos. Column sketches: A scan accelerator for rapid and robust predicate evaluation. In *SIGMOD*, 2018.
[57] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, 2007.
[58] I. Ilyas et al. Cords: Automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, 2004.
[59] Z. G. Ives and N. E. Taylor. Sideways information passing for push-style query processing. In *ICDE*, 2008.
[60] H. Kimura et al. Correlation maps: a compressed access method for exploiting soft functional dependencies. In *VLDB*, 2009.
[61] A. Lamb et al. The vertica analytic database: C-store 7 years later. *VLDB*, 2012.
[62] H. Lang et al. Data blocks: Hybrid oltp and olap on compressed storage using both vectorization and compilation. In *SIGMOD*, 2016.
[63] V. Leis et al. How good are query optimizers, really? In *VLDB*, 2015.
[64] A. Y. Levy, I. S. Mumick, and Y. Sagiv. Query optimization by predicate move-around. In *VLDB*, 1994.
[65] Y. Lu, A. Shanbhag, A. Jindal, and S. Madden. AdaptDB: Adaptive partitioning for distributed joins. In *VLDB*, 2017.
[66] I. S. Mumick and H. Pirahesh. Implementation of magic-sets in a relational database system. In *SIGMOD*, 1994.
[67] A. Nanda. Oracle exadata: Smart scans meet storage indexes. http://bit.ly/2ha7C5u, 2011.
[68] A. Nica et al. Statisticum: Data Statistics Management in SAP HANA. In *VLDB*, 2017.
[69] M. Olma et al. Slalom: Coasting through raw data via adaptive partitioning and indexing. In *VLDB*, 2017.
[70] C. Olston et al. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, 2008.
[71] J. M. Patel et al. Quickstep: A data platform based on the scaling-up approach. In *VLDB*, 2018.
[72] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *SIGMOD*, 1996.
[73] M. Saglam and G. Tardos. On the communication complexity of sparse set disjointness and exists-equal problems. In *FOCS*, 2013.

[74] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The uncracked pieces in database cracking. In *VLDB*, 2013.
[75] P. G. Selinger et al. Access path selection in a relational database management system. In *SIGMOD*, 1979.
[76] P. Seshadri et al. Cost-based optimization for magic: Algebra and implementation. In *SIGMOD*, 1996.
[77] A. Shanbhag et al. A robust partitioning scheme for ad-hoc query workloads. In *SOCC*, 2017.
[78] A. Shanbhag, A. Jindal, Y. Lu, and S. Madden. Amoeba: a shape changing storage system for big data. In *VLDB*, 2016.
[79] L. Shrinivas et al. Materialization strategies in the vertica analytic database: Lessons learned. In *ICDE*, 2013.
[80] J. Shute et al. F1: A distributed sql database that scales. In *VLDB*, 2013.
[81] D. Ślęzak et al. Brighthouse: An analytic data warehouse for ad-hoc queries. In *VLDB*, 2008.
[82] L. Sun et al. Fine-grained partitioning for aggressive data skipping. In *SIGMOD*, 2014.
[83] L. Sun, M. J. Franklin, J. Wang, and E. Wu. Skipping-oriented partitioning for columnar layouts. In *VLDB*, 2017.
[84] A. Thusoo et al. Hive- a warehousing solution over a map-reduce framework. In *VLDB*, 2009.
[85] N. Tran et al. The vertica query optimizer: The case for specialized query optimizers. In *ICDE*, 2014.
[86] B. Walenz, S. Roy, and J. Yang. Optimizing iceberg queries with complex joins. In *SIGMOD*, 2017.
[87] J. Yu and M. Sarwat. Two birds, one stone: a fast, yet lightweight, indexing scheme for modern database systems. In *VLDB*, 2016.
[88] M. Zaharia et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.
[89] E. Zamanian, C. Binnig, and A. Salama. Locality-aware partitioning in parallel database systems. In *SIGMOD*, 2015.
[90] H. Zhang et al. Surf: Practical range query filtering with fast succinct tries. In *SIGMOD*, 2018.
[91] J. Zhou et al. SCOPE: Parallel databases meet MapReduce. In *VLDB*, 2012.
[92] J. Zhou, P.-A. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the scope optimizer. In *ICDE*, 2010.

9 CONVERGENCE PROOF

We will now prove that the scheduling algorithm presented in §3.4 produces convergent schedules for tree-like (acyclic) join graphs. This proof makes no assumptions on the statistics used beyond those listed in §2; that is, statistics can identify satisfying partitions and are mergeable [33]. The proof assumes that merging statistics is not lossy; that is, the merged stat corresponding to a set of stats has the exact same information.

Recall from §3.4 that our schedule builds a tree for an acyclic join graph and passes data-induced predicates from the leaves of the join tree up to the root and then back down to the leaves. Our proof starts with some simple cases and builds towards the general case. We use the notation from Table 3 and slightly expand it to add epoch information. That is, $q_{t,(i)}$ denotes the partition subset of table t at the end of epoch i, and $d_{t\to r,(i)}$ denotes the data-induced predicate exchanged from table t to table r in epoch i.

One join: Suppose we have a single join between two relations, table t and table r, such as is 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 $d_{t\to r,(1)}$ and $d_{r\to t,(1)}$ 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 rows that are present in partition x.

On the one hand, partition x must have at least one row that satisfies 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) using $d_{t\to r,(1)}$, none of their join column values are contained in rows of table t that satisfy the local predicate on table t. Since, on the join column values, $x \subset S$, none of the rows in x can 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. We will show that none of these diPs can change a partition subset.

To see why the diP $d_{r\to s,(3)}$ cannot change the partition subset on s, apply the "one join" case above for the r–s join graph with the "local predicate" on tables `r` and `s` being p_r (as before) and $p_s \wedge d_{t\to s,(1)}$, respectively.

A similar argument applies for the other three diPs: $d_{s\to r,(3)}$, $d_{s\to t,(3)}$, 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" on tables `t` and `r` is p_t and p_s, respectively.

Hub and spoke join: Consider a join graph where a table h is the *hub* that any number of other tables, called *spokes*, join with. This generalizes both of the above join graphs which can be thought of as having either one or two spokes. It is easy to see that the schedule in §3.4 uses two epochs similar to the above cases; in the first epoch the spoke tables send diPs to the hub, and in the second epoch, the hub sends a diP to each spoke.

The proof of convergence follows from identical reasoning to the above. If there are n spoke tables, there are a total of $2n$ possible diPs, and we can show that none of these diPs can eliminate one more partition. The "local predicate" to use at a spoke is always that table's individual predicate, if any. The "local predicate" to use at the hub table h to prove a counter example for the diPs to or from a spoke s is $p_h \bigwedge_{t:t\neq s} d_{t\rightarrow h,(1)}$.

Arbitrary Tree: Recall that the schedule in §3.4 uses an "upwards pass" where data-induced predicates flow recursively from children to parents and a "downwards pass" in the opposite direction. The proof begins after both these passes have finished.

We will prove recursively by first considering the hub-and-spoke join graph consisting of the root and all of its children. Applying the logic from the hub-and-spoke case above, we can show that none of the diPs on any of the edges between the root and its children will change the partition subsets of these tables. To do so, we set the "local predicate" on each spoke node, i.e., child node of the root, as the conjunction of that child's local predicate, if any, and the data-induced predicates that the child receives from each of its children during the upwards pass of exchanging diPs. The "local predicate" on the root is similar to the case of the hub node above; i.e., to prove a counter example for the diPs to or from a child c, the "local predicate" at root r is $p_r \bigwedge_{t:t\neq c, t\in \text{child}(r)} d_{t\rightarrow r,(\text{up})}$; note that, instead of using the diPs from epoch (1) as above, this equation uses the diPs received by the root from its children during the "upwards pass".

Given this holds, we can repeat the same logic on each of the subtrees having a child as the root. Recursion stops at the leaf nodes.

Therefore, we can conclude that none of the diPs that can be exchanged along the tree edges will eliminate any partition.

9.1 Details of Treeify

The Treeify method takes as input an acyclic join graph and identifies the root node having the smallest tree height; parent – child pointers are also assigned along the way. A trivial tree can be constructed by picking *any node* as the root. However, we use Treeify to reduce the tree height because, as noted in §3.4, the smaller the tree height, the fewer the number of parallel epochs required for diP derivation to converge.

A trivial method to identify the smallest height tree takes $O(n^2)$ time for a join graph with n nodes because for each node as root, the height can be computed in $O(n)$ time.

We offer a simple algorithm that takes $2n$ time: pick a random node as the root r, do a depth first traversal on the tree to compute the tree height with r as the root; while doing the traversal, also record the height of each node. The height of a leaf node is 0 and the height of a node is one more than the maximum height of its children in the tree. In a second traversal over the tree built using the random root r, we iteratively consider whether picking any of the children of the current root will reduce tree height. Given the information computed during the first traversal, we can find the tree height for the case when some child c of the current root r becomes the new root; this tree height is equal to the maximum of the height of node c in the current tree and $1 + m$ where m is the maximum height of any other children of the current root r. If r has no other children, $m = 0$. If a switch reduces the tree-height, the only node whose height has to change is that of the current root r whose height becomes m. It is easy to see that this process will never reverse a switch (because tree height strictly decreases with each switch) and that the process considers each table exactly once.

Even though both Treeify and diP derivation over the tree require no more than $2n$ operations on a graph with n nodes, deriving and applying each diP, which is the operation involved during diP derivation, is much more complex than the simple comparisons and swaps needed for Treeify; thus, Treeify comprises a trivial fraction of the overall computation time.

9.2 Coping with cyclic join graphs

We first show a simple example illustrating the challenges with cyclic join graphs. Consider a three-way cyclic join $r \bowtie_a s \bowtie_b t \bowtie_c r$; here r, s, t denote three relations and a, b, c denote the equijoin columns. For this simple cyclic join, Table 8 illustrates a case which requires many diPs to be computed. In this case, only relation r has a local predicate which removes rows whose value for column a is a_0. In epoch #1, each relation exchanges diPs to all joining relations but only the diP from relation r to relation s cuts rows whose value of column a is a_0; assume that the corresponding values in column b are $\{b_0\}$. In epoch #2, only the diP from relation s to t cuts rows whose value of column b is b_0; suppose that the corresponding values in column c are $\{c_0\}$. In epoch #3, the diP from relation t to r cuts rows of r whose value of column c is c_0; assume that the corresponding values in column a are $\{a_1\}$. It is easy to see that this process continues with exactly one distinct value of a join column being pruned in each epoch. For this example, note that even a carefully constructed schedule for computing and exchanging diPs would not speed up convergence in any substantial manner. In the worst-case, the number of epochs (and diP computations) are bounded only by the minimum number of distinct values for the join columnsets. Note that this worst-case complexity can be much larger compared to

Relation r		Relation s		Relation t	
Row	**When/Why pruned**	**Row**	**When/Why pruned**	**Row**	**When/Why pruned**
$(a_0, .)$	local pred.	(a_0, b_0)	cut by $d_{r\to s,(1)}$	(b_0, c_0)	cut by $d_{s\to t,(2)}$
(a_1, c_0)	cut by $d_{t\to r,(3)}$	(a_1, b_1)	cut by $d_{r\to s,(4)}$	(b_1, c_1)	cut by $d_{s\to t,(5)}$
(a_2, c_1)	cut by $d_{t\to r,(6)}$	(a_2, b_2)	cut by $d_{r\to s,(7)}$	(b_2, c_2)	cut by $d_{s\to t,(8)}$

Table 8: Example illustrating convergence in a cyclic three-way join between relations r, s, t wherein the equijoin columns for $r \bowtie s$, $s \bowtie t$ and $t \bowtie r$ are a, b and c respectively.

the case of acyclic join graphs where the complexity depends only on the number of relations.

In practice, we find that convergence can be faster because diPs are computed over partitions which are large in size and hence fewer in number. Since convergence happens if an epoch does not prune at least one more partition in some relation, the total number of epochs is bounded by the total number of partitions which is significantly smaller than the above bound.

Nevertheless, the above counter-example convinces us to not aim for an optimal solution for cyclic join graphs. Recall from Algorithm 9 that we use an approximate approach that first constructs a tree-like graph over nodes that consist of connecting tables in the original join graph. We conjecture that such conversion to a tree-like graph over nodes does not sacrifice optimality and the ExchangeExt method also does not sacrifice optimality. In fact, optimality is likely lost in line 19 of the ProcessNode method where diPs are passed among relations within a node for only up to $\kappa = 5$ times. We leave proofs of these aspects to future work.

10 PROOFS RELATED TO RANGE-SETS

Given $\mathcal{X} = \{x\}$, a multi-set of column values, suppose that we want to construct a range-set of n_r ranges $\mathrm{RS} = \{[\ell_1, u_1], \ldots, [\ell_{n_r}, u_{n_r}]\}$ that covers all values in $\mathcal{X}$; i.e., for any $x_i \in \mathcal{X}$, there exists a range j such that $\ell_j \le x_i \le u_j$.

Lemma 1. *Splitting at the $n_r - 1$ largest* gaps *between contiguous values of $\mathcal{X}$ has the smallest width, defined as $\sum_{i=1}^{n_r}(u_i - \ell_i)$.*

Proof. If $\mathrm{RS}_{\mathrm{OPT}}$ is the range-set with the smallest width, we have

$$\mathrm{RS}_{\mathrm{OPT}} = \underset{\mathrm{RS}:|\mathrm{RS}|=n_r}{\operatorname{argmin}} \sum_{i=1}^{n_r}(u_i - \ell_i).$$

For this optimal range-set, the lower and upper values for the ranges will be $\ell_1 = \min(\mathcal{X})$ and $u_{n_r} = \max(\mathcal{X})$ respectively because if not, the width can be minimized further by setting the range limits to these values, contradicting that $\mathrm{RS}_{\mathrm{OPT}}$ is optimal. A similar reasoning could be used to show that every range boundary matches some value in $\mathcal{X}$ and that the ranges are non-overlapping, i.e., $u_i < \ell_j, \forall i < j$. Using these facts, with simple math, we can show

$$\min_{\mathrm{RS}:|\mathrm{RS}|=n_r} \sum_{i=1}^{n_r}(u_i - \ell_i) = u_{n_r} - \ell_1 + \max_{\mathrm{RS}:|\mathrm{RS}|=n_r} \sum_{i=1}^{n_r-1}(\ell_{i+1} - u_i).$$

Observe that $u_{n_r} - \ell_1$ is a constant and that each term in the sum on the right is the gap between consecutive ranges; hence, splitting the ranges at the $n_r - 1$ largest gaps is optimal. □

Lemma 2. *Given optimal range-sets $\mathrm{RS}(\mathcal{X}_i)$ for multi-sets of values $\mathcal{X}_i$, it is impossible to construct an optimal range-set which has the same number of ranges for the union of the multi-sets $\mathrm{RS}(\bigcup_i \mathcal{X}_i)$ (outside of a few special cases).*

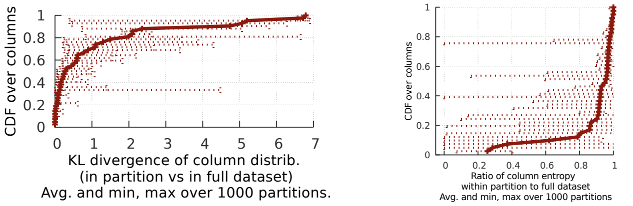
Proof. We offer a counter-example for $n_r = 2$.

$\mathcal{X}_1 = \{0, 11, 12, 14, 22\}$ $\quad \mathrm{RS}(\mathcal{X}_1) = \{[0, 0], [11, 22]\}$
$\mathcal{X}_2 = \{0, 4, 5, 10, 24, 25\}$ $\quad \mathrm{RS}(\mathcal{X}_2) = \{[0, 10], [24, 25]\}$
$\mathrm{RS}(\mathcal{X}_1 \cup \mathcal{X}_2) = \{[0, 14], [22, 25]\}$

The optimal range-sets shown on the right split the values at the largest possible gaps. Note that no possible method to merge the individual range-sets can achieve the optimal answer for the union because there is insufficient information to decide if and how the range $[11, 22]$ should be split. The large gap between 14 and 22 in the set $\mathcal{X}_1 \cup \mathcal{X}_2$ makes it the optimal split for that set, but notice that this information is not available in the individual ranges sets $\mathrm{RS}(\mathcal{X}_1)$ and $\mathrm{RS}(\mathcal{X}_2)$. Hence, merging already constructed range-sets is not optimal in general, but we note a few exceptions. First, trivially, if every set $\mathcal{X}_i$ has fewer than $2 * n_r$ distinct values, then each range-set $\mathrm{RS}(\mathcal{X}_i)$ fully captures all distinct values, no gap information is lost, and so merging is optimal. Next, if the multi-sets $\mathcal{X}_i$ are disjoint and non-overlapping, then merging their range-sets will be optimal because the gap *cut-off* (i.e., the smallest gap which leads to a range split) for the union range-set is at least as large as the gap cut-off of the individual range-sets. Finally, if one of the multi-sets, say $\mathcal{X}_a$, contains all of the distinct values in the union multi-set, then $\mathrm{RS}(\mathcal{X}_a)$ will contain every other range-set and is the optimal range-set of the union. □

Lemma 3. *Given two multi-sets that contain at least k distinct values each, consider the problem of sketching these sets so as to answer by just looking at the sketches whether their intersection is empty or not. The smallest possible sketch is at least of size $\theta(k \log k)$.*

Proof. The proof follows from observing that the above problem translates to the k-disjointness problem and applying known lower bounds (see [73] with number of rounds set to 1). □



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

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

Figure 23: Analysis of logs from production.

Notice that using data-induced predicates to skip partitions is akin to the set intersection problem above because we prune partitions on the destination table only if that partition's stat does not intersect with the diP from the source table. Hence, ensuring no false positives, i.e., to eliminate the maximum number of possible partitions, requires sketches that are roughly logarithmic in the number of distinct values. Key columns will have as many distinct values as the number of rows; it is common for join columns to be keys. Thus, sketches that will lead to maximal data skipping can be very large. Recall, that we only use a small constant number of ranges; thereby, we will have false positives and reduced data skipping gains but benefit from a smaller and maintainable statistic.

11 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 23a 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.[14] As further evidence, Figure 23b 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] $D_{KL}\left(Hist(c,p)||Hist(c)\right) = \sum_v -Prob_{c,p}(v)\frac{Prob_c(v)}{Prob_{c,p}(v)} \leq \ln(10^3) = 6.91$ 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 $\sum_p \texttt{Freq}_{c,p}(v) = \texttt{Freq}_c(v)$.

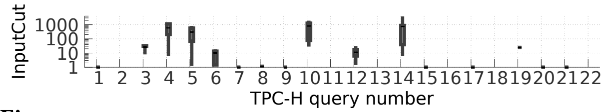
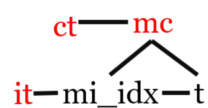


Figure 25: After adding a few changes, which we consider to be impractical, FineBlock can match the results presented in the original paper.

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



% 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 27: 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.

11.1 When will diPs give large gains?

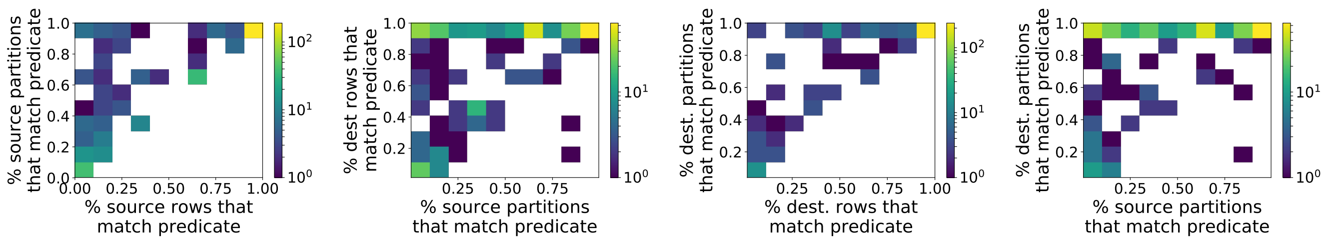
Following up on the description in §5.2, Table 9 lists which queries, predicates and data layouts satisfy some detailed conditions required for large gains from data-induced predicates.

11.2 Growth of false positives during construction and use of diPs

In Figure 24, 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 24(a), compares the fraction of rows filtered by a predicate with the fraction of partitions containing these rows. Note that the

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 9: 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 24: 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.

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 24(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 24(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 24(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 24(d) which will translate to InputCut= 2.).

11.3 Adaptive partitioning comparison

We mention a few additional details regarding our comparison with [82] which learns a clustering scheme over rows of a denormalized relation of TPC-H so as to enhance data skipping. We had to reimplement the algorithm in [82] because the code shared by the authors was missing some key pieces. We note some key aspects of our implementation, FineBlocks.

- As described in [82], we first partition rows of the denormalized relation shown in §14 by the month of `O_ORDERDATE` and then cluster together rows that match (or do not match) the same predicates, excluding date predicates.
- The authors of [82] 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.
- 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 [82] (note: this is 11, 000 partitions).
- The algorithm in [82] 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 §14 and hence are not run.

- 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 a 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 used by diPs. The time to skip partitions is also roughly similar; about 0.02s per query.

Our reimplementation of the algorithm from [82] 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 queries [82] such that almost all of the test predicates are available during training. These results are shown in Figure 25.

12 MORE END-TO-END EXAMPLES

Analogous to Figure 5, Figures 26 and 27 illustrate diPs in action for a query in TPC-DS [26] and in JOB [12], respectively. We choose these queries to illustrate how diPs work with complex statements (union operators, nested sql statements in TPC-DS q35 [27]) and cyclical joins (in JOB 1a [20]).

13 TUNED DATA LAYOUTS

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

13.1 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.

13.2 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.

14 DENORMALIZATION OF TPC-H

The following materialized view (or denormalized table) can support 16 out of 22 queries in the TPC-H benchmark [28]; 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.

```
CREATE TABLE 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
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
```

15 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.