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

Laurel Orr, Srikanth Kandula, Surajit Chaudhuri
Microsoft

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

## 1. INTRODUCTION

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

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

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

If only the predicate was on the column used in the join condition, _partkey, then a variety of techniques become applicable (e.g., algebraic equivalence [52], magic set rewriting [49, 74] or value-based pruning [83]), but predicates over

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

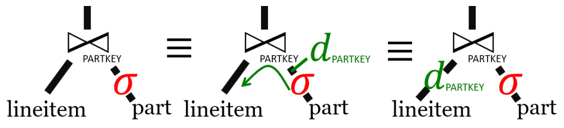


Figure 1: Example illustrating creation and use of a data-induced predicate which only uses the join columns and is a necessary condition to the true predicate, i.e., $\sigma \Rightarrow d_{\texttt{partkey}}$.

join columns are exceedingly rare[1] and these techniques do not apply when the predicates use columns that do not exist in the joining tables.

Some systems implement a form of sideways information passing over joins [19, 68] during query execution. For example, they may build a bloom filter over the values of the join column _partkey in the rows that satisfy the predicate on the part table and use this bloom filter to skip rows from the lineitem table. Unfortunately, this technique only applies during query execution, does not easily extend to multiple joins and has high overheads especially during parallel execution on large datasets because constructing the bloom filter becomes a scheduling barrier preventing the scan of lineitem from beginning before the bloom filter has been constructed.

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

Our target scenario is big-data systems, e.g., SCOPE [44], Spark [35, 87], Hive [81] or Pig [67] clusters that run SQL-like queries over large datasets; recent reports estimate upwards of a million servers in such clusters [1].

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

Using data statistics, we offer a method that converts predicates on a table to data skipping opportunities on the joining tables at query optimization time. The basic idea is

[1]Over all the queries in TPC-H [82] and TPC-DS [26], there are zero predicates on join columns perhaps because join columns tend to be opaque system-generated identifiers.

| Scheme | Statistic | Granularity |
|---|---|---|
| ZoneMaps [13] | max and min value per column | *zone* |
| Spark [8, 17] | | *file* |
| Exadata [64] | max, min and null present or null count per column | per table *region* |
| Vertica [58], ORC [2], Parquet [15] | | stripe, row-group |
| Brighthouse [78] | histograms, char maps per col | *data pack* |

Table 1: Data statistics maintained by several systems.

as follows; a simple example is in Figure 1. First, we use data statistics to eliminate partitions on tables that have predicates. This step is standard and is already implemented in some systems [6, 17, 44, 83]. Next, on the partitions that satisfy the local predicates, we use their data statistics to construct a new predicate which captures all join column values contained in these partitions. This new data-induced predicate (diP) is a necessary condition of the actual predicate (i.e., $\sigma \Rightarrow d$) because there may be false-positives; i.e., not all rows in the partitions included in the diP may satisfy $\sigma$. Also, note that the diP can apply over the joining table because it only uses the join column. Finally, we apply the diP on the data statistics of the joining table, `lineitem`, to eliminate partitions. All of these steps happen during query optimization; our QO effectively replaces each table with a partition subset of that table; the reduction in input size often triggers other plan changes (e.g., using broadcast joins which eliminate a partition-shuffle [46]) leading to more efficient query plans.

Note that if the above method is implemented using zone-maps, which are already maintained by many systems, 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 [45] or using join indexes [4, 22]. The fundamental reason is that these techniques use augmentary data-structures which are costly to maintain and yet their benefits are limited to narrow classes of queries (e.g., queries that match views, have specific join conditions, or very selective predicates) [33]. 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 [50, 62] or cluster rows that satisfy the same predicates [75, 79]; 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 [19, 68, 52, 83, 49, 74] or uses more complex structures which have sizable maintenance overheads [45, 4, 22, 79, 75]. To achieve the above, we offer an efficient method to compute diPs on complex query expressions (multiple joins, join cycles, nested statements, other operations). This method works with a variety of data statistics. We also offer a new statistic *range-set* that improves query performance over zone-maps at the cost of a small increase in space. We also discuss techniques to maintain statistics when datasets evolve. In more detail, the rest of this paper has these contributions.

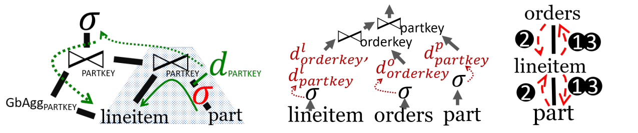


Figure 2: Illustrating the need to move diPs past other operations (left). On a 3-way join when all tables have predicates (middle), the optimal schedule only requires three (parallel) steps (right).

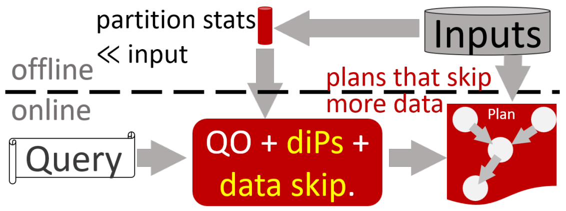


Figure 3: A workflow which shows changes in red; using partition statistics, our query optimizer computes data-induced predicates and outputs plans that read less input.

- Using diPs for complex query expressions leads to novel challenges. Consider TPC-H q17 which as shown in Figure 2(left) has two operations over the `lineitem` table. Creating diPs for only the fragment considered in Figure 1 still reads the entire `lineitem` table for the other operation. To alleviate this, we use new QO transformation rules to move diPs; in this case, shown with dotted arrows, the diP moves past two joins and a group-by and ensures that a single shared scan of `lineitem` will suffice. When multiple joining tables have predicates, a second challenge arises. Consider the 3-way join in Figure 2(middle) where all tables have local predicates. The figure shows four diPs one per table and per join condition. If applying these diPs eliminates any partition on a joining table, then the previously constructed diPs from that table are no longer up-to-date. Re-creating diPs whenever partition subsets change will increase data skipping gains but doing so naïvely can construct excessively many diPs which increases query optimization time. By establishing an analogy with inference on graphs, we present an optimal method for tree-like join graphs which converges to fixed point and hence achieves all possible data skipping while computing the fewest number of diPs. Joins in star and snowflake schemas are tree-like. Our solution for general join graphs and how we derive diPs within a cost-based query optimizer is in §3.
- We show how different data statistics can be used to compute diPs in §4 and discuss why *range-sets* represent a good trade-off between space usage, maintainability and potential for data skipping.
- We discuss two methods to cope with dataset updates

in §4.1. The first method *taints* a partition when any row in that partition changes; tainted partitions are never skipped; tables that contain tainted partitions cannot originate diPs but they can use diPs received from joining tables to eliminate untainted partitions. Our second method approximately updates data statistics by ignoring deletes and *growing* the statistic to cover new values. We will show in §5 that typical *range-sets* can be updated in tens of nanoseconds and that their usefulness decays gracefully as larger portions of the tables are updated.

- Fundamentally, data-induced predicates are beneficial only if the join column values in the partitions that satisfy a predicate contain only a small portion of all possible join column values. In §2.1, we discuss several real-world use-cases that lead to this property holding in practice and quantify their occurrence in production workloads .
- We report results from experiments on production clusters at Microsoft that have tens of thousands of servers. We also report results on SQL server. See Figure 3 for a high-level architecture diagram. Our results in §5 will show that using small statistics and a small increase in query optimization time, diPs offer sizable gains on three workloads (TPC-H [82], 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, $\sigma_{\texttt{d\_year}}(\texttt{date\_dim}) \bowtie_{\texttt{date\_sk}} \texttt{R}$. Table 2a shows the zone-maps per partition for the predicate and join columns. Recall that zone-maps are the maximum and minimum value of a column in each partition, and we use partition to denote the granularity at which statistics are maintained which could be a file, a rowgroup etc. (see Table 1). Table 2b shows the diPs corresponding to different predicates. The predicate column `d_year` is only available on the `date_dim` table but the diPs are on the join column `d_date_sk` and can be pushed onto joining relations using column equivalence [52]. The diPs shown here are DNFs over ranges; if the equijoin condition has multiple columns, the diPs will be a conjunction of DNFs, one DNF per column. Further details on the class of predicates supported, extending to multiple joins and handling other operators, are in §3.2. Table 2b also shows that the diPs contain a small portion of the range of the join column `date_sk` (which is $[1000, 12000]$); thus, they can offer large data skipping gains on joining relations.

It is easy to see that diPs can be constructed using any data statistic that supports the following steps: (1) identify partitions that satisfy query predicates, (2) *merge* the data statistic of the join columns over the satisfying partitions, (3) use the merged statistic to extract a new predicate and identify partitions that satisfy this predicate in joining relations. Many data statistics support these steps [29], 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

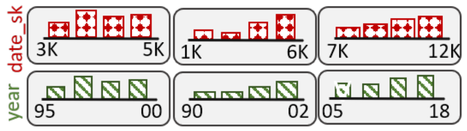
| **Column** | Partition # | | |
|---|---|---|---|
| | **1** | **2** | **3** |
| `d_date_sk` | $[3000, 5000]$ | $[1000, 6000]$ | $[7000, 12000]$ |
| `d_year` | $[1995, 2000]$ | $[1990, 2002]$ | $[2005, 2018]$ |

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

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

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

Table 2: Constructing diPs using partition statistics.



(a) Equiwidth histograms for the above example.

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

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

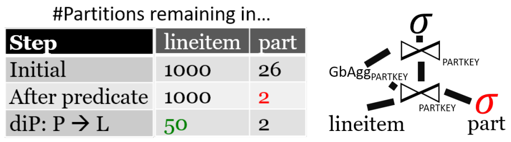
Figure 4: Showing other data statistics (histograms, range-sets) for the same example as in Table 2a.

with $b$ buckets uses $b+2$ doubles[2] compared to the two doubles used by zone maps (for the min. and max. value). Regardless of the number of buckets used, note that histograms will generate the same diPs as zone-maps. This is because histograms do not remember *gaps* between buckets. Other histograms (e.g., equi-depth, v-optimal) behave similarly. Moreover, the frequency information maintained by histograms is not useful here because diPs only reason about the existence of values. Guided by this intuition, consider a set of 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. Figure 4b shows range-sets of size 2. It is easy to see that range-sets give rise to more succinct diPs[3]. We will show that using a small number of ranges leads to sizable improvements to query performance in §5. We discuss how to maintain range-sets and why range-sets perform better than other statistics (e.g., bloom filters) in §4.

To assess the overall value of diPs, for the TPC-H query #17 from Figure 2(left), Figure 5 shows the I/O size reduction from using diPs. These results use a range-set of size 4 (i.e., 8 doubles per column per partition). The TPC-H dataset was generated with a scale factor of 100, skewed with a zipf factor of 2 [27], and tables were laid out in a

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

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



#Partitions remaining in...

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

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

typical manner[4]. Each partition is $\sim$ 100MBs of data which is a typical quanta in distributed file systems [38] and is the default in our clusters [90]. Even though the predicate columns are only available in the `part` table, the figure shows that only two partitions of `part` contain rows that satisfy the predicate, and the corresponding diP eliminates the vast majority of the partitions in `lineitem`. We will show results in §5 for many different data layouts and data distributions. We discuss plan transformations needed to move the diP, as shown in Figure 2 (left), in §3.3. Overall, for the 100GB dataset, a 0.5MB statistic reduces the initial I/O for this query by 20×; the query can speed up by more or less depending on the work remaining after initial I/O.

As an example of how the above conditions affect I/O savings, when the skew in the TPC-H dataset (generated using [27]) was reduced from a zipf factor of 2 to a zipf factor of 1.5 (or 1), the I/O savings for TPC-H query #17 reduce from 20× to 7.5× (or 1.02×). This is because lower skew weakens all three of the above conditions: the predicate on `part` is satisfied by rows in many more partitions of `part`; the corresponding diP spans a larger fraction of the value range of the join column `_partkey` and so on leading to reduced data skipping. Our results will show that across a variety of datasets, queries and predicates, all three of these conditions hold to a surprisingly large extent.

## 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 the diP only picks a small subset of partitions of the receiving table.

We identify use-cases where these conditions hold based on our experiences in production clusters at Microsoft [44].

[4]`part` was sorted on its key; `lineitem` was clustered on `l_shipdate` and each cluster sorted on `l_orderkey`; this layout is known to lead to good performance because it reduces re-partitioning for joins while allowing date-based predicates to skip partitions [3, 20, 24].

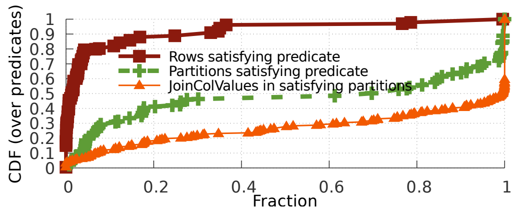


Figure 6: Quantifying how often the conditions that lead to large I/O skipping gains from using diPs hold in practice by using queries and datasets from production clusters at Microsoft.

- A majority of the datasets in production clusters are stored in the same order that the data was ingested into the cluster [35, 39]. A typical ingestion process consists of many servers uploading data in large batches. Hence, a consecutive portion of a dataset is likely to contain records for roughly similar periods of time and entries from a server are concentrated into just a few portions of the dataset. Thus queries for a certain time-period or for entries from a server will pick only a few portions of the dataset. This helps with `C1`.

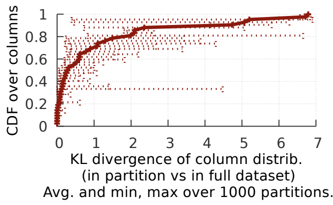
- A common physical design methodology for performant parallel plans is to hash partition a table by predicate columns and range partition or order by the join columns [3, 20, 24, 50]. Performance improves by reducing the shuffles needed to re-partition for joins [16, 91, 46] and by ensuring data skipping for predicates. Such data layouts help with all three conditions `C1`–`C3` and, in our experiments, translate to the largest I/O savings from diPs.

- Join columns are keys which monotonically increase as new data is inserted and hence are related to time. For example, both the title-id of movies and the name-id of actors in the IMDB dataset [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`.

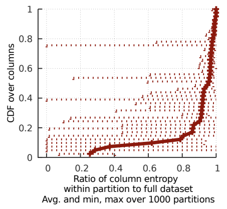
- Practical datasets are skewed; often times the skew is heavy-tailed [30]. When skew is large, both predicates and diPs can become more selective by skipping over heavy-hitters; hence, skew can help `C1`–`C3`.

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

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



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



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

Figure 7: Analysis of logs from production

To quantify the occurrence of such phenomenon, we analyzed logs from a large production data-parallel cluster at Microsoft. We analyzed the *in situ* layouts of crawled web snapshots, click logs and server logs. We analyze $10^3$ partitions for each dataset; the distributed file-system stores data as 100MB chunks. 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 7a 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.[5] As further evidence, Figure 7b 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.

## 3. CONSTRUCTION AND USE OF DATA-INDUCED PREDICATES

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

The algorithm has three building blocks: use predicates on individual tables to identify satisfying partitions, construct diPs for pairs of joining tables and apply diPs to further restrict the subset of partitions that have to be read on each table. Using the notation in Table 1, these steps can be written as:

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

Table 3: Notation used in this paper.

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

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

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

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

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

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

A second challenge arises when other relational operators can interfere with the simple method described above. That is, if the query expression consists only of select and join operations, the above method suffices. But, typical query expressions contain many other operations such as group-bys and nested statements. One option is to ignore other operations and apply diPs only to 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 restricting diPs to just the select-join portion (shown with a shaded background in the figure) may lead to no gains if the whole of `lineitem` table is read for the group-by. To address this challenge, we move diPs around other relational operators using commutativity. We list transformation rules in §3.3 that cover a broad class of operators. Using these transformation rules extends the usefulness of diPs to more complex query expressions.

### 3.1 Deriving diPs within a cost-based QO

[5] $D_{KL}\left(Hist(c,p)||Hist(c)\right) = \sum_v -Prob_{c,p}(v)\frac{Prob_c(v)}{Prob_{c,p}(v)} \leq \ln(10^3) = 6.91$ 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 $\sum_p \texttt{Freq}_{c,p}(v) = \texttt{Freq}_c(v)$.

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

Let's consider some alternative designs. (1) Could a query rewriter, separate from the QO, insert optimal diPs into the query? Then, the QO only needs minimal changes. This option is problematic because such a query rewriter has to implement complex logic that is already available within the QO. For example, the rewriter will have to implement predicate simplifications to identify parts of a query predicate that can apply on each table (the $p_i$'s in Eq. 1); as well, the rewriter will have to implement logic to move diPs around other operators (e.g., the rules in §3.3). (2) Can derivation of diPs be implemented as new plan transformation rules? If possible, this would be a small software change because the QO framework can remain unchanged. Unfortunately, diPs are sometimes exchanged multiple times between the same pair of tables and to keep costs manageable diPs have to be constructed in a careful order over the join graphs; in today's cost-based optimizers, such recursion and fine-grained query-wide ordering are challenging to achieve [52]. 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-by's, projections, predicates that use columns from multiple tables etc. diPs are exchanged first within these maximal select-join sub-portions of the query expression using the schedule in §3.4. Next, diPs are exchanged with the rest of the query using the rules in §3.3. With this method, derivation will be faster when the select-join sub-portions are large because by decoupling the above steps, we avoid propagating diPs which have not converged to other parts of the query. We also support one-sided outer joins (rule#5 in §3.3) and semi, anti-semi joins which are akin to set intersection and set difference respectively (rule#4 in §3.3).

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

We illustrate this overall process using an example.

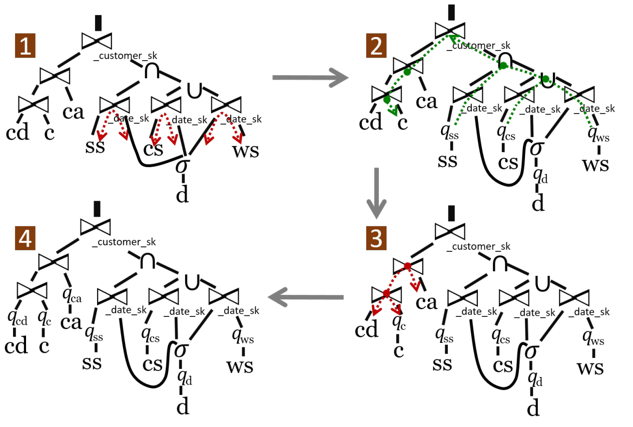


Figure 8: Illustrating deriving predicates for TPC-DS query 35. The table labels `ss`, `cs`, `ws` and `d` correspond to the tables `store_sales`, `catalog_sales`, `web_sales` and `date_dim`.

**Example:** Figure 8 shows this process in action for TPC-DS query 35; the large sql statement is in [?]. Data-induced predicates are computed first for each maximal SJ expression `store_sales` ⋈ `date_dim`, `catalog_sales` ⋈ `date_dim` and `web_sales` ⋈ `date_dim`, as shown in the top left of the figure labeled 1. As the plan shows, the result of these expressions joins with another expression on the `customer_sk` column after a few set operations. Hence, in 2, we build new diPs for the `customer_sk` column and pull those up through the set operations (union and intersection translate to logical or and and over diPs) and push down to the `customer` ($c$) table. To do so, we use the transformation rules in §3.3. In 3, if the incoming diP skips partitions on the `customer` table, another derivation on an SJ expression ensues.[6] The final plan, shown in 4, effectively replaces each table with the corresponding partition subset vector that has to be read from that table.

## 3.2 Supported Queries

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

**Predicates:** Our prototype triggers diPs for predicates which are conjunctions, disjunctions or negations over the following clauses using columns from a table:

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

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

**Projections:** On columns that are used in the above join conditions and predicates, only invertible projections (e.g.,

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

$\pi(x) = ax + b$ for column $x$ where $a, b$ are constants) commute with diPs on that column because only such projects can be inverted though zone-maps and other data statistics that we use to compute diPs. Arbitrary projections are supported on other columns.

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

## 3.3 Commutativity of data-induced predicates with other operations

We list some query optimizer transformation rules that apply to data-induced predicates (diPs). The correctness of these rules follows from considering a diP as a filter on join columns. In some cases, the alternatives are costlier (because they use redundant predicates) but we use them to facilitate movement of diPs. As noted in §3.1, diPs do not remain in the query plan; the diPs directly on tables are replaced with a read of the partition subsets of that table and other diPs are dropped.

1. diPs commute with any select.
2. A diP commutes with any projection that does not affect the columns used in that diP. For projections that affect columns used in a diP, commutativity holds if and only if the projections are invertible functions on one column.
3. diPs commute with a group-by if and only if the columns used in the diP are a subset of the group-by columns.
4. diPs commute with set operations such as union, intersection, semi- and anti semi-joins, as shown below.
    - $d_1(\mathcal{R}_1) \cap d_2(\mathcal{R}_2) \equiv (d_1 \wedge d_2)(\mathcal{R}_1 \cap \mathcal{R}_2) \equiv (d_1 \wedge d_2)(\mathcal{R}_1) \cap (d_1 \wedge d_2)(\mathcal{R}_2)$
    - $d_1(\mathcal{R}_1) \cup d_2(\mathcal{R}_2) \equiv (d_1 \vee d_2)(d_1(\mathcal{R}_1) \cup d_2(\mathcal{R}_2))$
    - $d(\mathcal{R}_1) - \mathcal{R}_2 \equiv d(\mathcal{R}_1 - \mathcal{R}_2) \equiv d(\mathcal{R}_1) - d(\mathcal{R}_2)$
5. diPs can move from one input of an equijoin to the other input if the columns used in the diP match the columns used in the equi-join condition. For outer-joins, a derived predicate can skip only if from the left side of a left outer join (and vice versa). No skipping is possible for a full outer join.
    - $d_c(\mathcal{R}_1) \bowtie_{c=e} \mathcal{R}_2 \equiv d_c(\mathcal{R}_1 \bowtie_{c=e} \mathcal{R}_2) \equiv d_c(\mathcal{R}_1) \bowtie_{c=e} d_e(\mathcal{R}_2)$; note here that $c$ and $e$ can be sets of multiple columns, then $c = e$ implies set equality.
6. As we saw in Figure 8 2, diPs on an inner join can push onto one of its input relations; generalizing the latter half of the above rule this requires the join input to contain all columns used in the diP, i.e., $d(\mathcal{R}_1 \bowtie \mathcal{R}_2) \equiv d(d(\mathcal{R}_1) \bowtie \mathcal{R}_2)$ iff all columns used by the diP $d$ are available in the relation $\mathcal{R}_1$.

To see these rules in action, note that the diP movement in Figure 8 2 uses rule#4 twice to pull up past a union and an intersection, rule #5 to move from one join input to another at the top of the expression and uses rule#6 twice to push to a join input. The example in Figure 2 (left) uses rule #5 at the above join and rule #3 to push below the group-by.

## 3.4 Scheduling the deriving of predicates

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

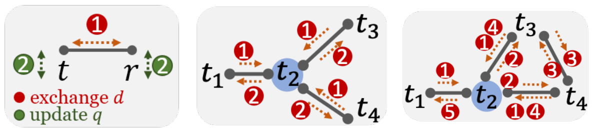


Figure 9: The optimal schedules of exchanging diPs for different join graphs; numbers-in-circles denote the epoch; multiple diPs are exchanged in parallel in each epoch. Details are in §3.4.

To build intuition, consider the examples in Figure 9. The simple case of two tables only requires a single exchange of diPs followed by an update to the partition subsets $q$; proof is in [10]. Next, the join graph in the middle with four tables in a star join requires six diPs in two (parallel) epochs. Any schedule where diPs *leave* $t_2$ before diPs from every neighbor go *into* $t_2$ will take more epochs or construct more diPs because $t_2$'s partition subset stabilizes only after receiving diPs from all of its neighbors. Finally, the join cycle graph on the right also has four tables but is slower to converge requiring ten diPs and five epochs. The key insight here is that some partition in each table may be eliminated only by the joint effect of all of the neighboring diPs and the local predicate but cannot be eliminated by any sub-combination of these predicates. Here, partition subsets stabilize first for tables $\{t_3, t_4\}$ at the end of third epoch because by then they receive all pertinent information. Notice also that the schedule avoids constructing diPs that are not needed for convergence (e.g., $t_4$ is silent in epochs $2, 5$) and constructs diPs on some edges multiple times (twice on the $t_3 \rightarrow t_2$ edge).

Our algorithm to hasten convergence is shown in Pseudocode 10, line#17, Scheduler method. For ease of exposition, consider the case of a join graph that has no cycles. This is an important sub-case because it applies to queries with star or snowflake schema joins. Here, we construct an implicit tree over the graph (Treeify in line#19), pass diPs up the tree (lines#7–#9) and then down the tree (lines#10–#12). To see why this converges, note that before line#10 executes, the partition subsets of the table at the root of the tree would have stabilized; Figure 9 (middle) illustrates this case with $t_2$ as root. For a join graph with $n$ tables, note that this method computes at most $2(n-1)$ diPs (because a tree has $n-1$ edges) and requires $\theta(\mathsf{depth}(\mathcal{G}))$ epochs where tree depth can vary from $\lceil \frac{n}{2} \rceil$ to $\lceil \log n \rceil$.

For join graphs that are not trees, we only briefly sketch our method for brevity. When the size of the largest clique in the join graph is small, we use a modified version of the junction tree algorithm [69]; see line#19 in Figure 10. Intuitively, this algorithm replaces each clique with a new virtual node and adds edges to retain connectivity as before. The process recurses until no cliques remain. Then, the above tree scheduler can be used with the caveat that receiving and constructing diPs at a virtual node require exchanging diPs among the tables that correspond to this node. Details are laborious but Figure 9 (right) illustrates an example where the clique $t_2, t_3, t_4$ can be thought of as a single virtual node. The complexity of the modified junction tree algorithm increases with the size of the largest clique, and hence, we fall back to a modified version of an approximate inference algorithm (CycleScheduler, line#15 in Figure 10)

```
Inputs: $\mathcal{G}$, the join graph and $\forall i, q_i$ denoting partitions
  to be read in table $i$ (notation is listed in Table 3)
Output: $\forall$ tables $i$, updated $q_i$ reflecting the effect of
  data-induced predicates
1  Func: DataPred $(q, \{c\})$ // Construct diP for columns
     $\{c\}$ over partitions $x$ having $q^x$ =1; see §4.
2  Func: Satisfy $(d, x)$ // = 1 if partition $x$ satisfies
     predicate $d$; 0 otherwise. See §4.
3  Func: Exchange$(i, j)$ : //send diP from table $i$ to table $j$
4  $d_{i\to j}$ = DataPred$(q_i$, ColsOf$(p_{ij}, t))$
5  $\forall$ partition $x \in$ table $j, q_j^x = q_j^x *$ Satisfy$(d_{i\to j}, x)$
6  Func: TreeScheduler$(\mathcal{T}, \{q_i\})$: // a tree-like join graph
7  for $h \leftarrow 0$ to height$(\mathcal{T}) - 1$ // bottom-up traversal do
8      foreach $t \in \mathcal{T}$ : height$(t) == h$ do
9          Exchange($t$, parent$(t, \mathcal{T})$)
10 for $h \leftarrow$ height$(\mathcal{T})$ to 1 // top-down traversal do
11     foreach $t \in \mathcal{T}$ : height$(t) == h$ do
12         $\forall$ child $c$ of $t$ in $\mathcal{T}$, Exchange$(t, c)$
13 Func: JunctionTree $(\mathcal{G}, \{q_i\})$
14 // construct a tree-like graph consisting of virtual nodes
     that represent sub-graphs; see [84]
15 Func: CycleScheduler $(\mathcal{G}, \{q_i\})$
16 // mimic approximate inference; see [84].
17 Func: Scheduler$(\mathcal{G}, \{q_i\})$:
18 if IsTree$(\mathcal{G})$ then return TreeScheduler
     (Treeify$(\mathcal{G})$, $\{q_i\}$);
19 if LargestClique$(\mathcal{G}) < \kappa$ then return JunctionTree
     $(\mathcal{G}, \{q_i\})$;
20 return CycleScheduler$(\mathcal{G}, \{q_i\})$
```

Figure 10: Pseudocode to compute a fast schedule.

when cliques are large [69]. In general, more complex join graphs require more epochs and exchange more diPs before partition subsets stabilize.

## 4. USING DATA STATISTICS TO BUILD diPs

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

**Zone-maps [13]** consist of the minimum and maximum value per column per partition. This stat uses 8 bytes per column per partition and is maintained by several systems today (see Table 1). Each predicate clause listed in §3.2 translates to a logical operation over the zone-maps of the columns involved in the predicate. Conjunctions, disjunctions and negations translate to an intersection, an union or set difference respectively over the partition subsets that match each clause. Typically, zone-maps store hashes for strings, and so equality check is also a logical equality, but regular expressions are not supported.

Note that there can be many false positives because a zone map has no information about which values are present

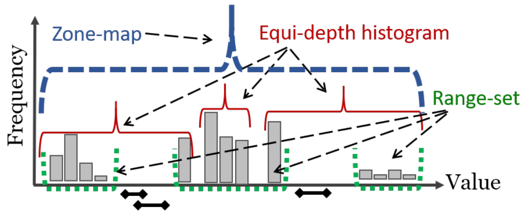


Figure 11: Illustrating the difference between range-sets, zone-maps and equi-depth histogram; both histograms and range-sets have three buckets. The predicates shown in black dumbels at the bottom of the figure will be false positives for all stats but the range-set.

(except for the min and max value).

The diP constructed using zone-maps, as we saw in the example in Table 2b, is a union of the zone-maps of the partitions satisfying the predicate; hence, the diP is a disjunction over non-overlapping ranges. On the table that receives a diP, a partition will satisfy the diP only if there is an overlap between the diP and the zone-map of that partition. Note that there can be false positives in this check as well because no actual data row may have a value within the range that overlaps between the diP and the partition's zone map. It is straightforward to implement these checks efficiently and our results will show that zone-maps can lead to sizable I/O savings (see Figures 12c and 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** [47] 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 [28, 51, 47], are expensive to maintain [51] and, the frequency information in histograms while very useful for other purposes is a waste of space here because predicate satisfaction and diP construction only check for existence of values.

**Bloom filters** record set membership [37]. However, we found them to be less useful for our purpose because the partition sizes used in practical distributed storage systems (e.g., $\sim 100MB$s of data [38, 90]) result in millions of distinct values per column in each partition, especially for join columns which are keys. Recording such large sets requires large space or will have a high false positive rate; for e.g., recording a million distinct values in a 1KB bloom filter has a 99.62% false positive rate [37] and almost no data skipping.

Alternatives such as the count-min [48] and AMS [34] 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) but as we noted in the case of histograms, frequency information is not helpful to construct diPs.

**Range-set:** To reduce false-positives while keeping the stat size small, we propose storing a set of non-overlapping ranges over the column value, $\{[l_i, u_i]\}$. Note that a zone-map is a range-set of size 1; using more ranges is hence a simple generalization. The boundaries of the ranges can be chosen to reduce false positives by minimizing the total width (i.e.,

$\sum_i u_i - l_i$) while covering all of the column values. To see why range-sets help, consider the range-set shown in green dots in Figure 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 increases the likelihood of false positives. Constructing a range-set over $r$ values takes $O(r \log r)$ time[7]. Reflecting on how zone-maps were used for the three operations in Equations 1– 3, i.e., applying predicates, constructing diPs and applying diPs on joining tables, note that a similar logic extends to the case of a range-set. SIMD-aware implementations can improve efficiency by operating on multiple ranges at once. A range-set having $n$ ranges uses $2n$ doubles. Merging two range-sets as well as checking for overlap between two range sets uses $O(n \log n)$ time where $n$ is the size of larger rangeset. Our results will show that small numbers of ranges (e.g., 4 or 20) lead to large improvements over zone-maps (Figure 16).

## 4.1 Coping with data updates

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

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

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

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

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

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

Taint bits can be maintained at transactional speeds and can be extremely effective in some cases, e.g., when updates are mostly in tables which do not generate data-reductive diPs. One such scenario is queries over updateable *fact* tables that join with many unchanging dimension tables; predicates on dimension tables can generate diPs that flow unimpeded by taint on to the fact tables. Going beyond one taint bit per partition, maintaining taint bits at a finer granularity (e.g., per partition and per column) can improve performance but with a small increase in update cost. **[pointer to result is a para]** Taint bits do not suffice, i.e., they will sacrifice I/O savings, if the tables that have query predicates (and which will originate data-reductive diPs) are updateable; for such cases, we propose a different method below that *grows* the data statistics.

**Approximately updating range-sets in response to updates:** The key intuition of this method is to update the range-set in the following approximate manner: ignore deletes and *grow* the range-set to cover the new values; that is, if the new value is already contained in an existing range there is nothing to do but otherwise either grow an existing range to contain the new value or collapse two existing ranges and add the new value as a new range all by itself. Since these options increase the total width of the range-set, the process greedily chooses whichever option has the smallest increase in total width. Table 4 shows examples of greedily growing a *range-set*. Our results will show that such an update is fast (Table 8) and the reduction in I/O savings, because the range-sets after several such updates can have more false positives than range-sets that are re-constructed for just the new column values, is small (Figure 18).

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

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

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

We also have some 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 [10].

## 5. EVALUATION

Using our prototypes in Microsoft's production data-parallel SCOPE clusters and SQL server, we aim to answer the following questions.

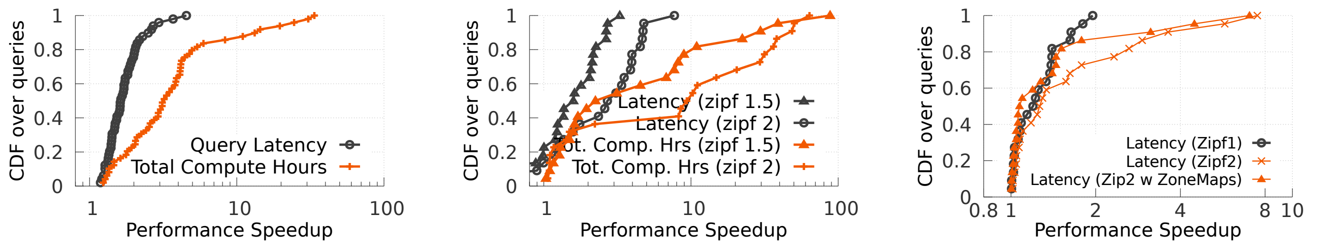
- Do data-induced predicates offer sizable gains on a large fraction of queries?
- Understand the causes for gains; that is, under what conditions will there be large gains?
- Understand the gap from alternatives.
- Estimate room for improvement and sensitivity to design choices, parameters etc.

We show that using diPs to eliminate partitions of tables reduces initial I/O by 1.5× in the median case; i.e, half of the queries from TPC-H, TPC-DS or the Join Order Benchmark on half of the data layouts read less than 1.5× their input. These savings in initial I/O improve query performance; in the data-parallel cluster, with diPs, the median query is 2× faster and uses 10× fewer resources. **[update result highlights]**

## 5.1 Methodology

**Workloads:** We report results on three public benchmarks: TPC-H [82], TPC-DS [26] and the join order benchmark (JOB) [60]. We use all 22 queries from TPC-H. TPC-DS and JOB have many more queries, and so we pick a subset of 50 queries from TPC-DS ($1 \ldots 40, 90 \ldots 99$) and 37 queries from JOB (([1−9]|10)*). Almost every query in JOB has join cycles which, as we saw in §3.4, can require careful

[7]First sort the values, then sort the gaps between consecutive values to find a cutoff such that the number of gaps larger than cutoff is at most the desired number of ranges; see [10] for proof of optimality.



(a) TPC-DS on Cosmos (b) TPC-H (skew=zipf 1.5 or 2) on Cosmos (c) TPC-H (skew=zipf 1.0) on SQL server

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

scheduling of derived predicates. Both the TPC benchmarks mimic decision support queries; however, -DS has more complex queries with several non foreign-key joins, UNIONs and nested SQL statements. Query predicates are complex; for example, q19 from TPC-H has 16 clauses over 8 columns from multiple relations. While inner-joins dominate, the queries also have some self-joins and outer joins.

For TPC-H and TPC-DS we generate 100GB and 1TB datasets respectively. The default datagen for TPC-H, unlike that of TPC-DS, creates uniformly distributed datasets which are known to not be representative of practical datasets, so we also use a modified datagen [27] to create datasets with different amounts of skew (e.g., with zipf factors of $1, 1.5, 2$). For JOB, we use the IMDB dataset from May 2013 [60].

**Systems:** We have built our prototype on top of two production platforms, SCOPE/ Cosmos clusters which serve as the primary platform for batch analytics at Microsoft and comprise tens of thousands of servers [44, 90] and SQL Server 2016. Both systems use cost-based CASCADE-style query optimizers [52]. A SCOPE job runs as a collection of tasks orchestrated by a job manager; tasks read and write to a file system but 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 interafce cards. Our SQL server experiments ran on a similar, single server. Cosmos uses a partitioned row store; for SQL server, we use columnstores which perform better for decision-support queries. Cosmos and SQL server have several advanced optimizations such as semijoin optimizations [19], predicate pushdown to eliminate partitions [6] and some magic-set rewrites [49].

**Data layouts and partitioning:** We report results on different layouts for each dataset. The tuned layout speeds up queries by avoiding re-partitioning before joins and enhances data skipping. [8] Our results will show that diPs yield sizeable gains on top of those accruing from the tuned layout. 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 corresponding storage layer, i.e., roughly 100MB of content in Cosmos [44, 90] and roughly 1M rows per columnstore segment in SQL Server [7].

[8]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 [10].

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

Table 5: The InputCut from diPs for different benchmarks; each query and data layout (see §5.1) contribute a point and the table reads out values at various percentiles. A quarter of the cases in TPC-DS receive an InputCut$> 4\times$; i.e., they only read $\frac{1}{4}$'th of the full input.

**Comparisons:** By Preds, we refer to a system that matches our prototype in all ways (i.e., uses same stats and skips partitions) except for using data-induced predicates. That is, Preds uses predicates to skip partitions on individual tables.

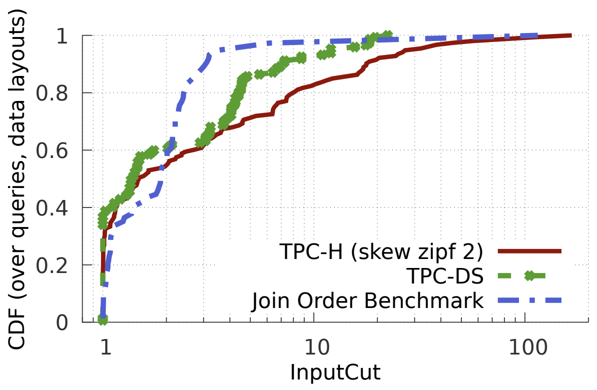
Since diPs improve data skipping gains for queries with joins, we compare against techniques that avoid joins by *denormalizing* the relations, i.e., materialize a single join view statement over multiple tables. By DenormView, we refer to a system that stores this view in column store format in SQL server. Queries on DenormView are over a single relation and so can use Preds to skip partitions without worrying about joins.

By FineBlock, we refer to a single-relation workload-aware partitioning system which enhances data skipping; it colocates rows that match or do-not-match the same predicates in the workload [79]. We apply FineBlock on the above denormalized view. The authors of [79] shared their code and we reimplement some missing parts.

**[Join indices?]**

**Statistics:** Many systems already store zone-maps, i.e., maximum and minimum value of each column in each partition, as noted in Table 1. We evaluate the value of using diPs for zone-maps and for other statistics mentioned in §4: *range-sets* maintain up to 20 min and max values and *hists* use up to 10 bucket discontinuous equi-depth histograms per column and per partition.

**Metrics:** Our primary metrics are query performance (latency and resource usage), statistic sizes, maintenance overheads, and increase in query optimization time. Since diPs reduce the input size that a query reads from the store, we also report the 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 $= 2$. Note that InputCut is always above 1 and is independent of the query execution engine. When comparing techniques or systems, we report the ratio of their metric values.



| 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) many queries have INPUTCUT> 4×; i.e., they only read $\frac{1}{4}$'th of the full input.

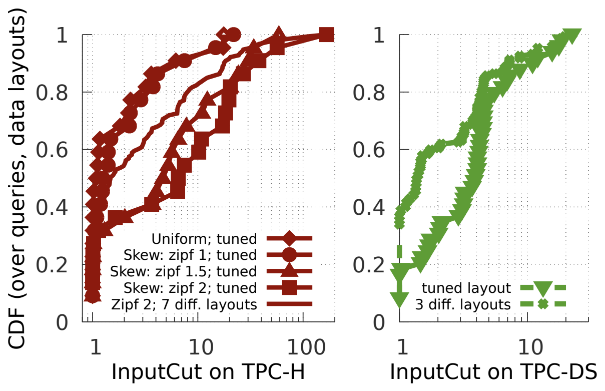


Figure 14: Input skew and data layouts affect the INPUTCUT from using data-induced predicates; see §5.2.

| | TPC-H | TPC-DS | JOB |
|---|---|---|---|
| Input size | 100GB | 1TB | 4GB |
| #Tables,#Columns | | | |
| # Queries | 22 | 50 | 37 |
| *Range-set* size | ∼ 2MB | ∼ 35MB | ∼ 3MB |
| # Partitions | $\sim 10^3$ | $\sim 4 * 10^4$ | $\sim 2 * 10^4$ |

Table 6: Additional results for experiments in Figure 12, Figure 13 and Figure 14. The table shows data from our Cosmos experiments; the range-set statistic is much smaller than the input. TPC-DS and JOB have more tables and more columns relative to TPC-H and hence need larger statistics.

| | 10th | 25th | 50th | 75th | 90th |
|---|---|---|---|---|---|
| **Baseline QO** | 0.145 | 0.158 | 0.176 | 0.188 | 0.218 |
| to add diPs | 0.032 | 0.050 | 0.084 | 0.107 | 0.280 |

Table 7: Showing different percentiles of the additional latency to derive diPs compared to the latency of the baseline query optimization time; see §5.2.

## 5.2 How much do derived predicates help?

We first report performance speed-up from using diPs in Cosmos and SQL server for the tuned layout which is a popular layout because it avoids re-partitioning for joins and enhances data skipping (see §5.1). Using diPs, Figure 12 shows that almost all queries improve. Cosmos is a shared cluster and so query latency is subject to substantial performance variability; the figures show the median value over at least five trials but still some TPC-H queries have a slight regression in latency. The figure on the left shows that the median TPC-DS query finishes almost 2× faster and uses 4× fewer total compute hours. Many queries receive (much) larger speed-ups. Typically, queries use many fewer total compute hours than their improvement in latency (higher speed-up in orange lines than in grey lines) because the changes in parallel plans that result from reductions in initial I/O can add to the length of the critical path which increases query latency while dramatically reducing total resource use; for e.g., replacing pair joins with broadcast joins eliminates shuffles but require merging the smaller input before broadcast [46]. The figures in the middle and right show that TPC-H queries receive similar improvemens in both Cosmos and SQL server. Unlike TPC-DS and real-world datasets which are skewed, the default datagen in TPC-H yields uniformly distributed datasets; these figures also show results with different amounts of skew generated using [27]. We see that diPs produce larger speed-ups as skew increases primarily because predicates and diPs become more selective at larger skew. The figure on the right shows sizable latency improvements in SQL server as well. Both the production systems, Cosmos and SQL server, implement a variety of optimizations related to predicate pushdown and semijoins [6, 19, 49]; these figures show that diPs offer sizable gains for a sizable fraction of benchmark queries on top of such optimizations.

We also evaluate diPs on many different data layouts as noted in §5.1 in addition to the tuned layout; Figure 13 shows a CDF of the savings in initial I/O (INPUTCUT from §5.1) across queries and data layouts. Figure 13 shows that at least 40% of the queries in each benchmark obtain an INPUTCUT of at least ∼ 2×; that is, they can skip over half of the input. The fraction of instances that receive at least an order of magnitude I/O reduction (x=10) is 2%, 5% and 19% in JOB, TPC-DS and TPC-H respectively.

Figure 14 teases apart the INPUTCUT values per layout and per skew. Lower skew leads to a lower INPUTCUT, but diPs offer sizable gains even for a uniformly distributed dataset (compare lines with different points in the figure on the left). The tuned data layout in both TPC-H and TPC-DS lead 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 ensures that diPs are succinct and can eliminate partitions on the receiving tables. **[Fact tables have multiple join columns and so data layouts that cluster by different join columns offer more speed-ups to some queries than tuned.]**

The cost to obtain this speed-up include storing statistics and an increase in the query optimization duration to determine which partitions can be skipped; Table 6 and Table 7 show these costs. Our derivation of diPs is not at production quality; it is a research prototype, parts of which are in c# for ease-of-debugging. Table 7 shows that the median addition to QO time is small; the outliers occur when diPs are exchanged between large fact tables because such diPs have many clauses and, on the receiving table, are evaluated on many partitions. However, note that evaluating diPs is embarassingly parallelizable; e.g., one thread per partition on receiving table. The range-set statistic can be thought off as roughly $n_r = 20$ "rows" per partition; a partition is 100MB of data in Cosmos and 1M rows in a columnstore segment in SQL server [7]; the space overhead for stats is hence $\sim 0.002\%$ [9].

These results, indicating that large performance speed-ups resulting from large fractions of inputs being skipped by using diPs for a sizable fraction of complex queries with only a small statistics make us optimistic about the usefulness of data-induced predicates.

## 5.3 Understanding why diPs help

We next try to understand the reasons behind these speed-ups. To obtain large speed-ups from data-induced predicates notice that all six of the following conditions have to be (at least partially) satisfied: (D1) the query has $> 1$ relation and predicates on individual relations, (D2) these predicates are selective, (D3) rows selected by predicates are concentrated in only a few partitions, (D4) the data statistic identifies skippable partitions, and (D5) join column values from the unskippable partitions of a relation are concentrated in only a few partitions of the joining relation. We assess how often these conditions are met for each query in TPC-H next; Figure 15 shows the consequent InputCut values. Table 9 shows additional detail.

- Gains are wide-spread: 15/22 queries receive large I/O savings; namely {**q2**, **q3**, **q4**, **q5**, q6, q7, **q10**, **q12**, q14, q15, **q16**, **q17**, **q19**, q20, **q21**}
  - In 10/22 queries, shown in **red** above, pushing data-induced predicates past joins skips much more data than using predicates on individual tables (shown as Preds in Figure 15). For these queries, diPs magnify the gains from predicate pushdown; predicates on small relations can eliminate many partitions on the large relations.
  - In the remaining queries, diPs only modestly improve upon `Preds`. This is because:
    - One query has no joins {q6}.
    - The other four queries have selective predicates on the largest table, and so diPs do not offer much additional data-skipping.
- Among the 7/22 queries that receive little or no speed-up, {q1, q8, q9, q11, q13, q18, q22},
  - One query has no predicates; {q18}.
  - Four have predicates that are not selective; {q1, q9, q13, q22}.
  - Two have rows matching the predicate spread across almost all partitions; {q8, q11}.

To summarize, diPs will yield large InputCut when the conditions D1–D6 are met. Evidence from experiments illustrates that these conditions hold often; although, different data distributions and layouts produce different amounts of gains.

[9] Recall that we use partition, colloquially, to refer to the input granularity at which statistics are stored.

| 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 |

Table 8: Average and standard deviation of the time to greedily update range-sets of various sizes measured in nanoseconds on a desktop.

## 5.4 Effect of various design choices

**Gains with different statistics:** Figure 15 shows the InputCut when using different statistics. Notice that zone-maps (the max. and min. value per column per partition) are often as good as histograms (we use equi-depth histograms with 10 buckets and gaps here) to construct diPs; compare the third blue candlestick in each cluster with the second green candlestick. The figure also shows that the range-set statistic, the first red candlestick in each cluster, is better than both zone-maps and histograms for the purposes of constructing diPs (recall that we use 20 ranges per column per partition).

**Gains with different number of ranges:** Figure 16 shows that just 4 ranges per partition achieve nearly the same amount of data skipping as much larger range-sets. This is because XXX.

**Comparing different methods to construct diPs:** Figure 17 shows that the algorithm presented in §3.4 leads to more succinct diPs compared to a naïve method that constructs the same number of diPs but in a random order. For a few queries, the join graph is very simple and the naïve algorithm suffices but for a strong majority of the queries, careful choice of the order of constructing diPs substantially improves data skipping.

**Gains when tainting partitions:** Recall from §4 that we proposed to taint partitions that receive data updates; tables with tainted partitions cannot originate diPs, but diPs from other relations can eliminate partitions that are not tainted. We evaluate this approach by using the TPC-H data generator to generate 100 different update sets each of which change 0.1% of the `orders` and `lineitem` tables. Of the 15/22 queries where diPs deliver sizable gains (see Figure 15 and §5.3), 5 queries remain unaffected; specifically {q2, q14, q15, q16, q17, q19}. That is, diPs lead to large I/O savings on these queries in spite of updates. Since updates in TPC-H target the two largest tables, `lineitem` and `orders`, both relations become tainted and diPs cannot flow *from* of these relations, and so any query that requires a diP out of these tables loses InputCut due to taints. As noted in §4, taints are better suited when updates are targeted at smaller dimension tables.

**Greedily maintaining range-sets:** Recall from §4 that our second proposal to cope with data updates is to greedily grow the range-set statistic to cover the new values. Table 8 shows that range-sets can be updated in tens of nanoseconds using one core on a desktop; thus, the anticipated slowdown to a transaction system is negligible. Our experiments also show that the greedy update procedure leads to a reasonably high quality rangset 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. Figure 18 shows that the greedy updates lead to a range-set with an average gap value that is over 80% of that of the

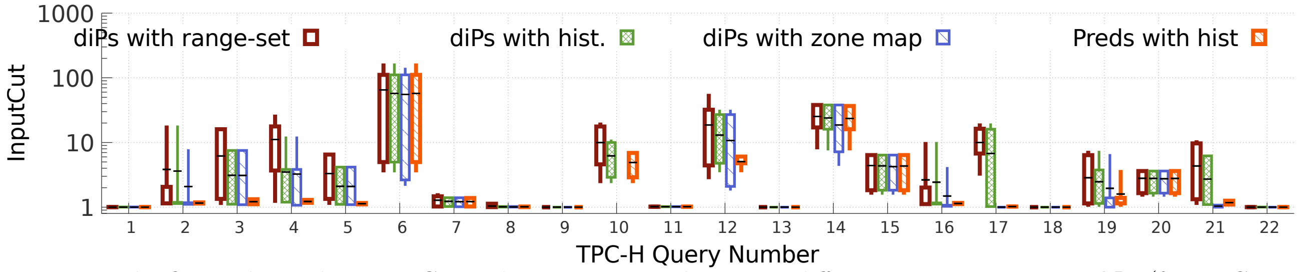


Figure 15: The figure shows the InputCut values per query when using different stats to construct **diPs** (for TPC-H with zipf 2 skew); InputCut values from standard predicate pushdown are also shown. Candlesticks show variation across seven different data layouts including the **tuned** layout; the 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 but range-sets are a sizable improvement.

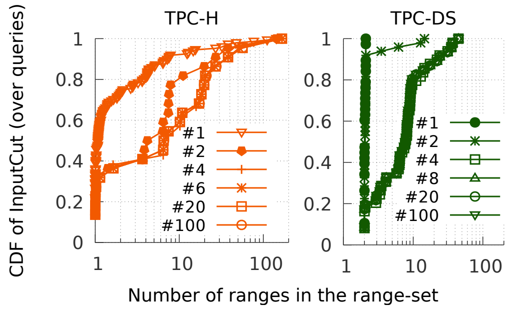


Figure 16: The figure shows how InputCut varies with the numbers of ranges used in the range-set statistic. (Results are for TPC-H skewed with zipf 2 and TPC-DS in the **tuned** layout; other cases behave similarly). Using just four ranges achieves most of the InputCut.

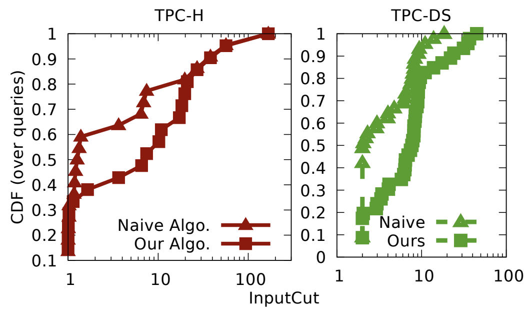


Figure 17: The figure shows how InputCut varies when different methods are used to derive **diPs**; we compare our algorithm from §3.4 with a naïve algorithm that constructs the same number of **diPs**. (Results are for TPC-H skewed with zipf 2 and TPC-DS in the **tuned** layout; other cases behave similarly).

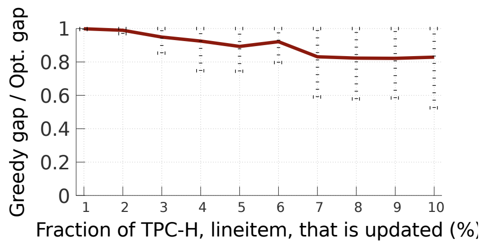


Figure 18: Greedily updating the *range-set* stat when different fractions of the `lineitem` table in TPC-H are updated; the figure shows the average and stdev across columns.

optimal range-set when up to 10% of rows in the `lineitem` table are updated.

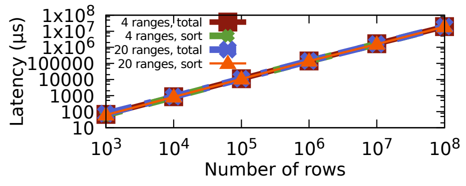


Figure 19:

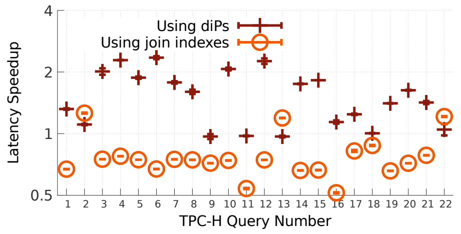


Figure 20:

**Range set construction time:** Figure 19 shows the latency to construct range-sets; as the figure shows, computing larger rangesets (e.g., 4 ranges vs. 20 ranges) has only a small impact on latency and almost all of the latency is due to sorting the input once (the 'total' lines are indistinguishable from the 'sort' lines). The results here use std::sort method from Microsoft Visual C++. Note that range-sets can be constructed during data ingestion; the latency shown here is well within the throughput of many ingestion pipleines. Also, note that range-set construction can piggy-back on the first query to scan the input.

**Join Indexes:** Figure 20 compares the query latency obtained from using **diPs** on SQL server with those from using join indexes. Results are for TPC-H skewed with zipf factor 1 with a scale factor of 100. We built clustered indexes [5] on the key columns of the dimension tables and on the fact tables we built clustered indexes on their columns that are most frequently used in join conditions (i.e., `l_orderkey`, `o_orderkey`, `ps_partkey`). Indexes are not supported on columnstores; so we use rowstores for just this experiment. The figure shows that join indexes lead to worse query latency than using default SQL server without indexes; we believe this is due to the following reasons: (1) the predicate selectivity in TPC-H queries is often not small enough to benefit from an index seek and so most plans use a clustered index scan to read the input relations and (2) clustered index scans are slower than table scans. Using **diPs** instead of indexes leads to no improvement for

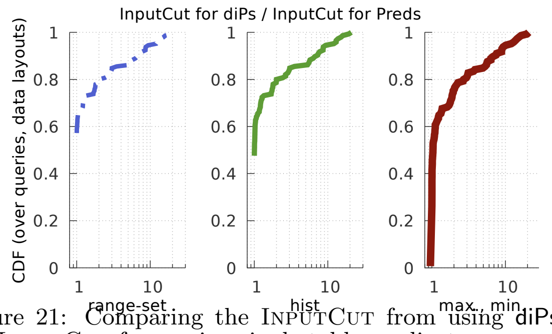


Figure 21: Comparing the InputCut from using diPs vs the InputCut from using single-table predicates on queries from TPC-H over all 7 data layouts; we see that diPs skip more partitions.

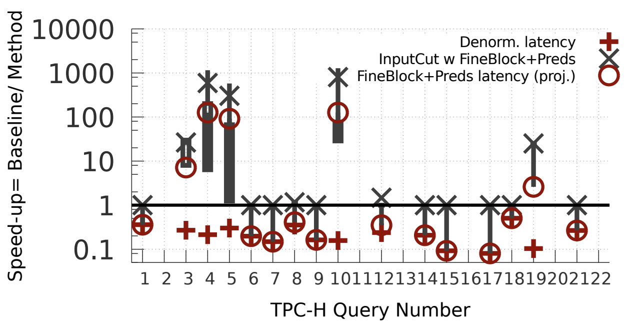


Figure 22: InputCut on TPC-H queries for FineBlock+Preds; box plots contain results for different predicates per query.

a few queries where diPs are unable to offer additional data skipping but most of the queries receive a sizable speed-up.

## 5.5 Comparing with alternatives

**Data-induced predicates vs. just predicates:** Figure 21 shows the ratio of improvement over Preds, a technique that skips partitions only on individual tables. We see that data-induced predicates are a marked improvement over single-table data skipping. As noted above, diPs do not yield much additional gains over Preds for queries that have no joins or have selective predicates only on large relations.

**Data-induced predicates vs. denormalization:** A materialized view (denormalized relation) is shown in §13 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. This view in columnar (rowstore) format occupies 2.7× (6.2×) more storage space than all of the other tables combined. Queries that can use this view are un-hindered by joins because all predicates directly apply on a single relation; however, because the relation is larger in size, queries may or may not finish faster. 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) and some of the queries slow down by over 10×.

**Data-induced predicates vs. clustering rows by predicates:** A recent research proposal [79] clusters rows in the above view (denormalized relation) to maximize data skipping. Using a training set of query predicates, [79] learns a clustering scheme over rows that is intended to improve data skipping for unseen queries and predicates. Figure 22 shows with $x$ symbols the average InputCut obtained as a result of such clustering; the candlesticks around the $x$ symbol show the min, 25th percentile, 75th percentile and max InputCut for different query predicates. We see that most queries receive no InputCut ($x$ marks are at 1) due primarily to two reasons: (1) the chosen clustering scheme does not generalize across queries; that is, while some queries and predicates receive large data skipping, the average benefits are small, 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 scheme. We see that 5/22 queries improve (fastest query is ∼ 100× faster) while 11/22 queries regress (slowest query is 10× slower). Hence, it is unclear that such a scheme is practically useful. We also note the rather large overheads to learn the clustering [79], to maintain the view [31, 70] and the unclear generalizability of these techniques to complex queries and complex predicates. In contrast, diPs only use small and easily maintainable data statistics and can offer sizable improvements for many more queries. Also, diPs require no apriori knowledge of queries (to select appropriate views [33, 32] or to identify row clusters [79]) and can help with both ad-hoc and complex queries.

## 6. RELATED WORK

While there has been much research on data partitioning, statistics, and join optimizations, to the best of our knowledge, this 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, they are built on-the-fly in the QO and skip partitions of multiple joining relations even in complex queries; the resulting plans only read subsets of the input relations and have no execution-time overhead.

Some research works discover data properties and use them to improve query plans, e.g., functional dependencies and column correlations [9, 41, 55, 57]. Inferring such data properties has a sizable overhead (e.g., [55] uses student t-test between every pair of columns). Also, imprecise data properties are less useful for QO (e.g., a *soft* functional dependency unlike exact functional dependency does not preserve set multiplicity and hence cannot guarantee correctness of certain plan transformations over group-bys and joins). A SQL server option [9] uses the fact that the `l_shipdate` attribute of `lineitem` is between 0 to 90 days larger than the `o_orderdate` attribute of `orders` in TPC-H [82] to convert predicates on `l_shipdate` to predicates on `o_orderdate` and vice versa. Some works discover similar constraints more broadly [41, 57]. In contrast, diPs do not infer any data properties; rather, they exploit relationships that may only hold conditionally given a predicate and a data-layout. For example, 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 conditions happen when data layouts are clustered on time (e.g., seal log partitions in the order that data is uploaded) or when tables are partitioned on join columns [50] irrespective of the underlying data properties.

Prior work that moves predicates around joins relies on column equivalence and magic-set style reasoning [49, 61, 63, 74, 83, 85]. Both Cosmos and SQL server implement some of these optimizations and we saw in §5 that diPs offer gains on top of these baselines. Prior techniques do not push predicates past joins because predicates are rarely on join columns and [74] shows that magic set transformations help only 2/22 of the TPC-H queries and only when predicates are very selective. This work has wider applicability because it builds diPs using data statistics.

Auxiliary data structures such as views [25], join indices [22], join bitmap indexes [4], succinct tries [89], column sketches [53], partial histograms [86] can also help speed-up queries. Constructing and maintaining these data structures has overhead and as we saw in §5 a particular view or join index does not help with all queries; diPs can be thought off as a simpler and complementary alternative that helps *tail* queries. Also, diPs can deliver more value if such auxiliary structures are available; for example, diPs can be applied during query execution as SARG-able predicates on indices.

While data-induced predicates are similar to the implied integrity constraints used by [65], they are some key differences and additional contributions. (1) We show broader applicability beyond the data-aging context considered by [65]. (2) Whereas [65] only exchanges constraints between a pair of tables, we offer a general method to exchange diPs between multiple relations. We also handle cyclic joins and queries having group-by's, union's and other operations. (3) Whereas [65] uses zone maps and two bucket histograms, we offer a new statistic (range-set) that performs better. (4) Whereas [65] shows no query performance improvements we show speed-ups in both a big-data system and a DBMS. (5) Whereas [65] offers no results in the presence of updates, we design and evaluate two maintenance techniques that can be added to transactional systems and show that the diPs offer gains even when large fractions of datasets are updated.

While a query executes, sideways information passing [36, 56, 68, 74, 77] (SIP) from one subexpression to a joining subexpression can prune the data-in-flight and speed up the query. Several systems including SQL server implement SIP and we saw in §5 that diPs offer additional speed-up. This is because SIP applies during query execution and, unlike using diPs, does not reduce the first read size. SIP can reduce the communication costs 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 and introduces an extra barrier that prevents simultaneous parallel computation of the joining relations. Also, unlike diPs, SIP cannot easily extend to the case of multiple joins nor does it create new predicates that can be pushed below group-by's, unions and other operations.

A large area of related work aims to improve data skipping using workload-aware adaptations to data partitioning (either offline [79, 80] or on-line [62, 66, 76]) or to indexing [62, 76, 88, 80]. These systems have different targets–streams [43], raw files [66], package queries [42], columnar stores [54, 72]—but share a key insight: co-locate data that is accessed together or build correlated indices to enhance data skipping. Some of these systems use views which denormalize the relations to avoid joins [79, 88]. In contrast, diPs require no changes to the data layout and no foreknowledge of queries, they work with whichever partitioning strategy is already used by the storage layers.

Oracle's join zone maps [14] on a table can include columns from other tables that can be outer-joined to this table; that is, a fact table can maintain join zone maps consisting not only of its columns but also of the predicate columns from dimension tables that can join with the fact table. This is similar to maintaining zone maps on (partially) denormalized relations; in contrast, we maintain statistics only for individual tables and support predicates on any column in any table by constructing diPs on-the-fly at query optimization time.

## 7. CONCLUSION

As dataset sizes grow, human-digestible insights increasingly use queries with selective predicates. In this paper, we show a new technique that extends the possible 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. We extend the query optimizer to output plans that skip data before the query begins (e.g., partition elimination). In contrast to prior work that offers complex auxiliary structures, workload-aware adaptation 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; the median case improves by about 1.5× (e.g., a typical TPC query on a typical data layout) and over 10× improvement occurs on a sizable fraction of cases. We are hopeful that data-induced predicates can be added to big-data stacks which already maintain required data statistics.

## 8. REFERENCES

[1] 2017 big-data and analytics forecast. `https://bit.ly/2TtKyjB`.
[2] Apache orc spec. v1. `https://bit.ly/2J5BIkh`.
[3] Apache spark join guidelines and performance tuning. `https://bit.ly/2Jd87We`.
[4] Bitmap join indexes in oracle. `https://bit.ly/2TLBBTF`.
[5] Clustered and nonclustered indexes described. `https://bit.ly/2Drdb9o`.
[6] Columnstore index performance: Rowgroup elimination. `https://bit.ly/2VFpljV`.
[7] Columnstore indexes described. `https://bit.ly/2F7LZuI`.
[8] Data skipping index in spark. `https://bit.ly/2qONacb`.
[9] Date correlation optimzation in sql server 2005 & 2008. `https://bit.ly/2VodSVN`.
[10] Imdb datasets. `https://imdb.to/2S3BzSF`.
[11] Join order benchmark. `https://bit.ly/2tTRyIb`.
[12] Oracle database guide: Using zone maps. `https://bit.ly/2qMeO9E`.
[13] Oracle: Using zone maps. `https://bit.ly/2vsUWKK`.
[14] Parquet thrift format. `https://bit.ly/2vm6D5U`.
[15] Presto: Repartitioned and replicated joins. `https://bit.ly/2JauYll`.
[16] Processing petabytes of data in seconds with databricks delta. `https://bit.ly/2Pryf2E`.

[17] Pushing data-induced predicates through joins in bigdata clusters; extended version. .

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[33] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala. Database tuning advisor for microsoft sql server 2005. 2004.

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

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

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

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

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

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

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

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

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

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

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

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

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

[47] 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.

[48] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. 2005.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[69] J. Pearl. *Probabilistic Reasoning in Intelligent*

*Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

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

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

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

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

[74] P. Seshadri et al. Cost-based optimization for magic: Algebra and implementation. In *SIGMOD*, 1996.

[75] A. Shanbhag et al. A robust partitioning scheme for ad-hoc query workloads. In *SOCC*, 2017.

[76] A. Shanbhag, A. Jindal, Y. Lu, and S. Madden. Amoeba: a shape changing storage system for big data. *VLDB*, 2016.

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

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

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

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

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

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

[83] N. Tran et al. The vertica query optimizer: The case for specialized query optimizers. In *ICDE*, 2014.

[84] M. J. Wainwright and M. I. Jordan. Graphical models, exponential families, and variational inference. 2008.

[85] B. Walenz, S. Roy, and J. Yang. Optimizing iceberg queries with complex joins. In *SIGMOD*, 2017.

[86] J. Yu and M. Sarwat. Two birds, one stone: a fast, yet lightweight, indexing scheme for modern database systems. *VLDB*, 2016.

[87] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[88] E. Zamanian, C. Binnig, and A. Salama. Locality-aware partitioning in parallel database systems. In *SIGMOD*, 2015.

[89] H. Zhang et al. Surf: Practical range query filtering with fast succinct tries. In *SIGMOD*, 2018.

[90] J. Zhou, N. Bruno, M.-C. Wu, P.-Å. Larson, R. Chaiken, and D. Shakib. SCOPE: parallel databases meet MapReduce. *VLDB J.*, 21(5):611–636, 2012.

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

## 9. DISCUSSION

**Other uses of diPs:** Some systems may apply diPs over an index [73] or at the remote store [21] and receive more data skipping than from partition elimination. Other systems may choose to execute diPs during query execution (e.g., when inputs are in memory); 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 on-the-fly semijoin optimizations [19].

**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 [59].

**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 $n_r * n_b$ 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 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:**

## 10. ADDITIONAL RESULTS

### 10.1 When will diPs give large gains?

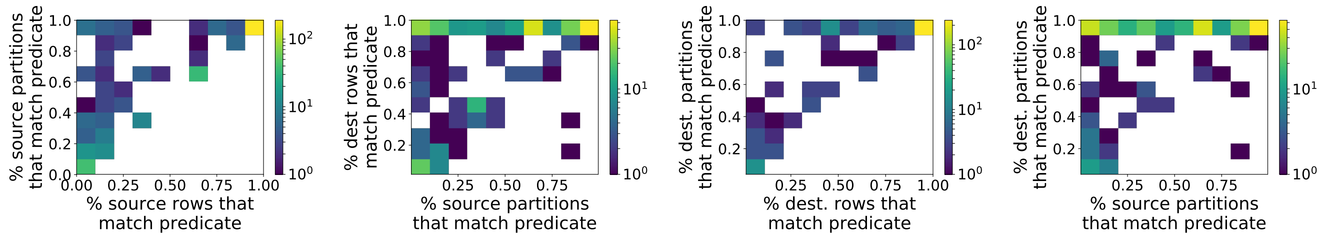
Following up on the description in §5.2, Table 9 lists which queries, predicates and data layouts satisfy the conditions required to obtain large gains from data-induced predicates.

### 10.2 Growth of false positives during construction and use of diPs

In Figure 23, 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 23(a), compares the fraction of rows filtered by a predicate with the fraction of partitions containing these rows. Note that the figures are 2d histograms on a logarithmic scale. We see substantial concentration on the $y = 1$ line indicating that many predicates, even those that are selective, may not filter out partitions. Figure 23(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 23(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 23(d) shows

| Condition | Queries that **do not** manifest this condition |
|---|---|
| C1: query has predicates on smaller relation(s) | **q18** |
| C2: 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) |
| C3: 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) |
| C4: stat can identify skippable partitions | regex: (q2, q9, q13) |
| C5: 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 23: 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.

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

## 10.3 Adaptive partitioning comparison

We mention a few additional details regarding our comparison with [79] which learns a clustering scheme over rows of a denormalized relation of TPC-H so as to enhanced data skipping. We had to reimplement the algorithm in [79] because the code shared by the authors was missing some key pieces. We note some key aspects of our implementation, FineBlocks. (1) As described in [79], we first partition rows of the denormalized relation shown in §13 by the month of O_ORDERDATE and then cluster together rows that match (or do not match) the same predicates, excluding date predicates. (2) The authors of [79] have also stated that they rewrote query predicates using hard-coded constraints between the L_SHIPDATE and O_ORDERDATE columns. Such constraints are not available in general across tables; hence, we do not use such rewrites in FineBlock. (3) The FineBlock results use a TPC-H scale factor of 1 because we had trouble scaling to larger dataset sizes; however, we scale down the minimum partition size to create the same number of partitions as in [79] (note: this is 11,000 partitions). (4) The algorithm in [79] is sensitive to training data and may not work well when the test data is very different from training because the rows are clustered only based on predicates that are available during training. We train FineBlock on 8 query templates with 30 queries each; namely $\{3, 5, 6, 8, 10, 12, 14, 19\}$. We test FineBlock on 16 query templates, 10 queries per template; namely $\{1, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 15, 17, 19, 20, 21\}$. The remaining 6 query templates in TPC-H are not contained in the denormalized relation shown in §13 and hence will receive no benefit from FineBlock+Preds. (5) The time to train the workload-aware partitioning and to re-layout the dataset is sizable (for a 1GB dataset, takes about 2400s, single-threaded, on an x86 linux server with 1TB memory); this process is compute bottlenecked and the time should increase with the number of rows; it should grow much more quickly if the dataset spills from memory. Storing the partitioning metadata of FineBlock requires somewhat less space than the data stats used by diPs; $3500B$ to maintain a dictionary of the predicates used as features for partitioning and roughly $10B$ per partition to store a bit vector of which features are matched by a partition versus about $2000B$ per partition for pdSkip. The time to skip partitions is also roughly similar; about $0.02s$ per query.

Our reimplementation of the algorithm from [79] 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 [79] such that almost all of the test predicates are available during training. These results are shown in Figure 24.

## 11. MORE END-TO-END EXAMPLES

Analogous to Figure 5, Figures 25 and 26 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

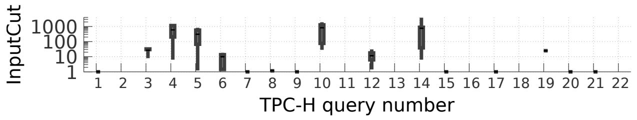


Figure 24: After adding a few changes, which we consider to be impractical, FineBlock can match the results presented in the original paper.

statements in TPC-DS q35 [23]) and cyclical joins (in JOB 1a [18]).

## 12. 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 $\texttt{RS} = \{[\ell_1, u_1], \ldots, [\ell_{n_r}, u_{n_r}]\}$ that covers all values in $\mathcal{X}$; i.e., for any $x_i \in \mathcal{X}$, there exists a range $j$ such that $\ell_j \leq x_i \leq u_j$.

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 $\texttt{RS}_{\text{OPT}}$ is the range-set with the smallest width, we have

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

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 $\texttt{RS}_{\text{OPT}}$ 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

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

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 $\texttt{RS}(\mathcal{X}_i)$ 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 multisets $\texttt{RS}\left(\bigcup_i \mathcal{X}_i\right)$ (outside of a few special cases).*

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

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

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 $\texttt{RS}(\mathcal{X}_1)$ and $\texttt{RS}(\mathcal{X}_2)$. 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 $\texttt{RS}(\mathcal{X}_i)$ 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 $\texttt{RS}(\mathcal{X}_a)$ 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 [71] with number of rounds equal to 1). □

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, to ensure 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.

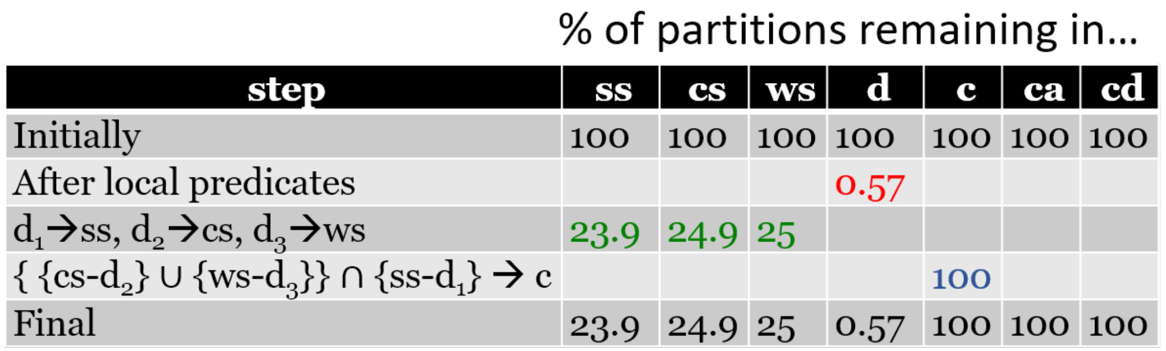
## 13. DENORMALIZATION OF TPC-H

The following materialized view (or denormalized table) can support 16 out of 22 queries in the TPC-H benchmark [82]; specifically, queries $\{2, 11, 13, 16, 20, 22\}$ cannot be answered using just this view because those queries require information that is absent in the view.

**CREATETABLE** denorm **AS**
**SELECT** lineitem.*, customer.*, orders.*, part.*, partsupp.*, supplier.*, n1.*, n2.*, r1.*, r2.*
**FROM** lineitem **JOIN** orders **ON** o_orderkey = l_orderkey
**JOIN** partsupp **ON** ps_partkey = l_partkey **AND** ps_suppkey = l_suppkey
**JOIN** part **ON** p_partkey = ps_partkey
**JOIN** supplier **ON** s_suppkey = ps_suppkey
**JOIN** customer **ON** c_custkey = o_custkey
**JOIN** nation **AS** n1 **ON** n1.n_nationkey = c_nationkey
**JOIN** nation **AS** n2 **ON** n2.n_nationkey = s_nationkey
**JOIN** region **AS** r1 **ON** r1.r_regionkey = n1.n_regionkey
**JOIN** region **AS** r2 **ON** r2.r_regionkey = n2.n_regionkey

## 14. HANDLING UPDATES TO DATASETS

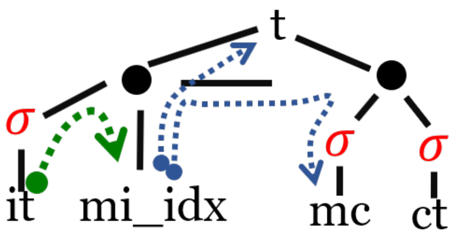
The primary use-case for diPs is data warehouses and big-data clusters where datasets are read-only or are appended to in large batches. In this cases, statistics can be constructed on newly arriving batches before making the data available to queries. We note that this is a widely prevalent use-case;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

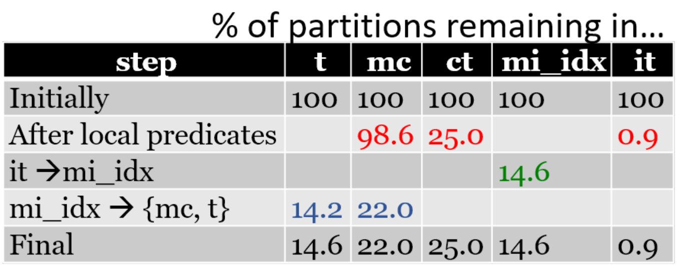


% of partitions remaining in…

| 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 25: 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 | 14.6 | 22.0 | 25.0 | 14.6 | 0.9 |

Figure 26: 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.

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 range-set? 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.

## 15. CONVERGENCE PROOF

We will now prove that the scheduling algorithm presented in subsection 3.4 produces convergent schedules. This proof makes no assumptions on the partition statistics used, as long as the statistics meet the criteria listed in section 2 (i.e. can identify satisfying partitions and are mergeable). The proof does, however, assume that no information is lost when merging statistics; i.e., no new false positives are introduced during diPs construction. The only false positives are from the inherent lossiness of using data statistics.

For ease of notation, let $S_i(p_i)$ return the entire $q_i$ vector for table $i$ from calling Satisfy$(p_i, x)$ for all $x$. Let $DP_i(q_i, p_{ij})$ denote DataPred$(q_i, p_{ij})$ being run on table $i$. Lastly, denote $\hat{q}_i$ as the *converged* partition vector for table $i$.

Note the following transformations, which follow from our assumption of no new false positives.

$$DP(q_i * q_i', p_{ij}) = DP(q_i, p_{ij}) \wedge DP(q_i', p_{ij}) \quad (7)$$

$$S(d_{i\to j} \wedge d_{i\to j}') = S(d_{i\to j}) * S(d_{i\to j}') \quad (8)$$

Let's first consider a join graph $\mathcal{G}$ that is a tree with two nodes, table 1 and table 2. Each table will converge once it has applied its local predicate and received the *converged* derived predicate from its neighbor. In other words,

$$\hat{q}_1 = S_1(p_1 \wedge DP_2(\hat{q}_2, p_{21}))$$
$$\hat{q}_2 = S_2(p_2 \wedge DP_1(\hat{q}_1, p_{21}))$$

since $p_{21} = p_{12}$.

Using the transformation rules, we get

$$\begin{aligned}\hat{q}_1 &= S(p_1) * S_1(DP_2(\hat{q}_2, p_{21}))\\ &= S_1(p_1) * S_1(DP_2(S_2(p_2 \wedge DP_1(\hat{q}_1, p_{21})), p_{21})).\end{aligned}$$

Expanding out the second term, we get
$S_1(DP_2(S_2(p_2 \wedge DP_1(\hat{q}_1, p_{21})), p_{21}))$

$$\begin{aligned}&= S_1(DP_2(S_2(p_2) * S_2(DP_1(\hat{q}_1, p_{21})), p_{21}))\\ &= S_1(DP_2(S_2(p_2), p_{21}) \wedge DP_2(S_2(DP_1(\hat{q}_1, p_{21})), p_{21}))\\ &= S_1(DP_2(S_2(p_2), p_{21})) * S_1(DP_2(S_2(DP_1(\hat{q}_1, p_{21})), p_{21})).\end{aligned}$$

The final subterm of $S_1(DP_2(S_2(DP_1(\hat{q}_1, p_{21})), p_{21}))$ is taking the converged $\hat{q}_1$ vector, sending it to table 2, and applying the new derived predicate to $\hat{q}_1$. However, by definition of $\hat{q}_1$ being the converged partition vector, this last step adds no new information to the partition vector of table 1. This means we can remove that step from the final equation without impacting $\hat{q}_1$. Therefore, we get

$$\hat{q}_1 = S_1(p_1) * S_1(DP_2(S_2(p_2), p_{21}))$$

where a similar statement holds for $\hat{q}_2$.

This indicates that when table 2 sends $DP_2(S_2(p_2), p_{21})$ to table 1, then table 1 can converge. Likewise for table 2.

Generalizing to any tree structure will the root being table $r$ and $\mathcal{N}(i)$ being the neighbors of table $i$, we get that...