Optimizing Disjunctive Queries with Tagged Execution

ALBERT KIM, MIT, USA
SAMUEL MADDEN, MIT, USA

Despite decades of research into query optimization, optimizing queries with disjunctive predicate expressions remains a challenge. Solutions employed by existing systems (if any) are often simplistic and lead to much redundant work being performed by the execution engine. To address these problems, we propose a novel form of query execution called *tagged execution*. Tagged execution groups tuples into subrelations based on which predicates in the query they satisfy (or don't satisfy) and tags them with that information. These tags then provide additional context for query operators to take advantage of during runtime, allowing them to eliminate much of the redundant work performed by traditional engines and realize predicate pushdown optimizations for disjunctive predicates. However, tagged execution brings its own challenges, and the question of what tags to create is a nontrivial one. Careless creation of tags can lead to an exponential blowup in the tag space, with the overhead outweighing the benefits. To address this issue, we present a technique called tag generalization to minimize the space of tags. We implemented the tagged execution model with tag generalization in our system Basilisk, and our evaluation showed an average 2.7× speedup in runtime over the traditional execution model with up to a 19× speedup in certain situations.

CCS Concepts: • **Information systems → Database query processing**.

Additional Key Words and Phrases: Query Optimization; Disjunctions; Tagging; Tagged Execution

ACM Reference Format:
Albert Kim and Samuel Madden. 2024. Optimizing Disjunctive Queries with Tagged Execution. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 158 (June 2024), 25 pages. https://doi.org/10.1145/3654961

1 INTRODUCTION

Despite decades of research in query optimization, optimizing queries with disjunctive predicates remains a relatively understudied problem. Many systems and optimizers ignore disjunctions outright, and solutions (if any) employed by existing systems often fallback to simple and inefficient heuristics for evaluating disjunctive queries. However, disjunctions continue to commonly appear in real workloads and pose issues for query optimization [8] [15] [1].

To illustrate the challenges, suppose we want to compile a list of potential movies to watch this weekend. We are a fan of more recent movies because they have better special effects, so we are willing to watch them as long they have a score above 7.0. However, we are also willing to tolerate the effects of older movies if they are "masterpieces" and have a score greater than 8.0. Query 1 (next page) expresses these constraints. Note the schema for this query comes from the IMDB dataset provided by the Join Order Benchmark [27], and the query is a simplification of one of the queries in the benchmark[1].

Due to the disjunction in Query 1's predicate expression, performing pushdown optimizations is not straightforward. In this case, existing systems typically do one of two things: (1) Perform the

[1]We take a few liberties with the attribute names and predicate expressions in this example to improve brevity and clarity.

Authors' addresses: Albert Kim, alkim@csail.mit.edu, MIT, Cambridge, MA, USA; Samuel Madden, madden@csail.mit.edu, MIT, Cambridge, MA, USA.

ACM 2836-6573/2024/6-ART158
https://doi.org/10.1145/3654961

```
SELECT * FROM title AS t JOIN movie_info_idx AS mi_idx
             ON t.id = mi_idx.movie_id
       WHERE (t.year > 2000 AND mi_idx.score > '7.0')
          OR (t.year > 1980 AND mi_idx.score > '8.0')
```

Query 1

join first, then evaluate the predicate expression on the resulting joined output. (2) Treat each part of the disjunction as a separate query, applying pushdown optimizations separately, and combine the results with a union operator. Note this is equivalent to internally transforming Query 1 into:

```
SELECT * FROM title AS t JOIN movie_info_idx AS mi_idx
             ON t.id = mi_idx.movie_id
       WHERE t.year > 2000 AND mi_idx.score > '7.0'
UNION
SELECT * FROM title AS t JOIN movie_info_idx AS mi_idx
             ON t.id = mi_idx.movie_id
       WHERE t.year > 1980 AND mi_idx.score > '8.0'
```

Both solutions are unsatisfactory. The first option is equivalent to performing no optimizations whatsoever. Query 1 only features a single join, but without a way to prune the relations before the join, as the number of joins grows, the size of the joined output will grow exponentially and the total runtime alongside with it. The second option allows for pushdown optimizations, but does redundant work because movies that scored above 8.0 and were made after 2000 satisfy both parts of the disjunction. The tuples representing these movies need to be constructed multiple times. An additional, potentially expensive union operator is also required to filter out duplicate tuples. It is also worth noting that the second option is only even available because Query 1's predicate expression is a disjunct of conjuncts (DNF). If it were a conjunct of disjuncts (CNF), the second option would not be available.

To address these problems, we propose a novel form of query execution called *tagged execution*. In traditional query execution, operators such as filters and joins operate on and result in sets of tuples called relations. In tagged execution, tags loaded with semantic information are attached to subsets of relations, and operators use that extra information to avoid performing any redundant work during query execution, thereby significantly improving runtime performance. The tagged execution model allows the query optimizer to push down disjunctive predicates regardless of whatever form the predicate expression might have. In addition, it ensures that each tuple is only ever materialized once and that each predicate subexpression is only evaluated once, even if it appears multiple times in the overall predicate expression.

Optimally executing Query 1 under tagged execution would:

(1) Apply predicates `t.year > 2000` and `t.year > 1980` to the `title` table and attach the tags **`{t.year > 2000 = T}`** and **`{t.year > 2000 = F, t.year > 1980 = T}`** to the relevant tuples.
(2) Apply predicates `mi_idx.score > '8.0'` and `mi_idx.score > '7.0'` to the `movie_info_idx` table and attach the tags **`{mi_idx.score > '8.0' = T}`** and **`{mi_idx.score > '8.0' = F, mi_idx.score > '7.0' = T}`** to the relevant tuples.
(3) Do the following joins between the sets of tuples with tags:
    (a) **`{t.year > 2000 = T}`** $\times$ **`{mi_idx.score > '8.0' = T}`**
    (b) **`{t.year > 2000 = T}`** $\times$ **`{mi_idx.score > '8.0' = F, mi_idx.score > '7.0' = T}`**
    (c) **`{t.year > 2000 = F, t.year > 1980 = T}`** $\times$ **`{mi_idx.score > '8.0' = T}`**

Note the predicates are pushed down, and the join only ever processes each tuple once.

Challenges. Although tagged execution is a powerful new paradigm for query execution, it introduces new technical challenges:

(1) **Tag Management.** How should tags be generated, which should be preserved, and how should they be combined? A naive implementation storing all true/false assignment values of predicates can lead to an exponential number of tags, so we must manage tags carefully to ensure overheads do not outweigh the benefits.
(2) **Planning.** A new execution model requires a new query planner. We must explore into how much of conventional planning wisdom we can bring into tagged execution and devise new planners which take advantage of the unique benefits offered by tagged execution.

Contributions. In short our contributions are:

(1) The tagged execution model, with its ability to optimize and push down disjunctive predicates.
(2) Our solution for the tag management problem and several new planners to use with tagged execution.
(3) The evaluation of tagged execution in our system Basilisk[2], in which tagged execution achieved an average 2.7× speedup in runtime over traditional query execution with up to a 19× speedup in certain situations.

The rest of this paper is structured as follows. Section 2 provides a detailed explanation of the tagged execution model. Section 3 introduces a technique called tag generalization to solve the tag management problem and discusses how to handle NULL values. Section 4 provides an overview into planning for tagged execution. Section 5 presents our evaluation, Section 6 discusses related work, and Section 7 concludes the paper.

2 TAGGED EXECUTION

In this section, we describe the tagged execution model. We introduce the concept of tags and describe how operators in tagged execution use these tags to avoid redundant work. It should be noted that the actual creation of tags and the decision of which tags to create are all done and made by the planner during plan time, which we describe in Section 3. This section only focuses on how the execution engine performs tagged execution during runtime, given a query plan and some set of tags.

As an overarching example for this section, consider Figure 1 (next page), which shows a sample query plan for the tagged execution of Query 1. The base table nodes `title` and `movie_info_idx` feed into the four base predicates from Query 1 before coming together for a final join. As we describe next, this plan successfully achieves disjunctive predicate pushdown. The orange numbers in parentheses refer to Examples 1 to 4 (Page 5), which list example tuples for each of these stages. Focusing on the left side, the tuples from `title` are originally assigned the empty tag (i.e., **`{}`**). After applying the predicate `t.year > 2000`, the tuples which evaluate to true are assigned the tag **`{t.year > 2000 = T}`**, and the tuples which evaluate to false are assigned the tag **`{t.year > 2000 = F}`**. The second filter can then avoid redundant work by applying the predicate `t.year > 1980` on only the tuples with the tag **`{t.year > 2000 = F}`**; the tuples with the tag **`{t.year > 2000 = T}`** can pass through the filter untouched. Note the second filter does not output the tuples which would be assigned the tag **`{t.year > 2000 = F, t.year > 1980 = F}`**. These tuples do not satisfy the overall predicate expression, so they can be dropped here and do not need to be fed into the join. After performing similar actions on the right side, the join operator then uses the tags to selectively join only the tuples which satisfy the overall predicate expression. In particular, the tuples which would

[2]The common basilisk can run across water by "pushing down" with its feet rapidly enough to create air pockets to push off of [17]!

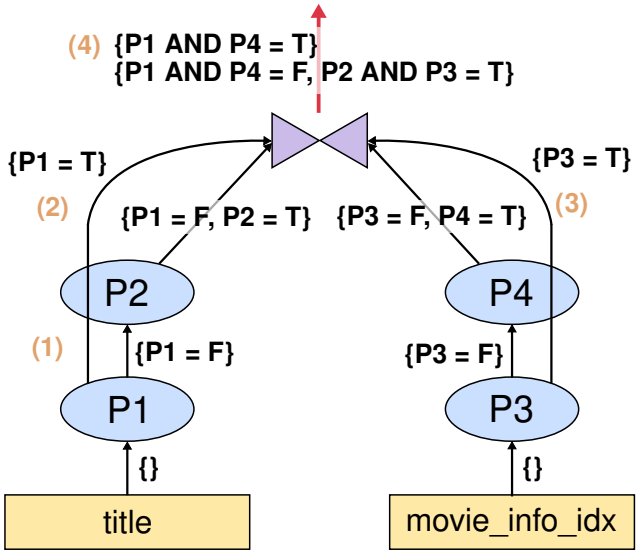


Fig. 1. Example tagged execution query plan of Query 1. Here, P1 = `t.year > 2000`, P2 = `t.year > 1980`, P3 = `mi_idx.score > '8.0'`, and P4 = `mi_idx.score > '7.0'`.

be associated with the tag **`{t.year > 2000 = F, t.year > 1980 = T, mi_idx.score > '8.0' = F, mi_idx.score > '7.0' = T}`** are never joined. Thus, by relying on tags to keep track of predicate results, tagged execution is able to avoid much of the redundant work that would be performed by the traditional execution model.

In the remainder of this section, we first define the concept of relations with tags in Section 2.1 and then describe how operators uses these tags in tagged execution in Sections 2.2 to 2.4. We wrap up with some important implementation details regarding tagged execution in Section 2.5.

2.1 Setup

In traditional query execution, operators process and produce sets of tuples, or *relations*. In tagged execution, this basic unit of operation becomes *tagged relations*. Tagged relations are similar to regular relations in that they comprise of sets of tuples, but subsets of tuples in tagged relations (called *relational slices*) are annotated with tags containing semantic information, providing additional context for operators to take advantage of during query execution. A tagged relation may have any number of relational slices. However, the relational slices must be mutually exclusive, and every relational slice must be associated with exactly one tag. Although the model allows for relational slices with zero tuples, in practice, these relational slices are removed from tagged relations for performance reasons. The tags themselves are sets of true/false *assignments* to arbitrarily complex predicate expressions from the query, and an assignment has the form:

$$\langle expr \rangle = T/F$$

In which $\langle expr \rangle$ is an arbitrarily complex Boolean SQL expression, and an assignment to T represents $\langle expr \rangle$ is true, and an assignment to F represents $\langle expr \rangle$ is false. Each tag may have any number of assignments, and each tuple in the corresponding relational slice must satisfy every assignment present in the associated tag. For example, every tuple in the relational slice associated with the tag **`{t.year > 2000 = F, t.year > 1980 = T}`** must be a title produced between 1981 and 2000.

Examples 1 to 4 are all examples of tagged relations associated with Query 1. The schema for the tuples are shown in the captions. Tags are in bold, and corresponding relational slices are given in

```
{t.year > 2000 = T}:
     {('The Dark Knight', 2008, 1), ('Evolution', 2001, 2), ('Avatar', 2009, 7)}
{t.year > 2000 = F}:
     {('The Shawshank Redemption', 1994, 3), ('Pulp Fiction', 1994, 4),
      ('The Godfather', 1972, 5), ('Beetlejuice', 1988, 6)}
```

Example 1. Pre-filter `title(title, year, id)`

```
{t.year > 2000 = T}:
     {('The Dark Knight', 2008, 1), ('Evolution', 2001, 2), ('Avatar', 2009, 7)}
{t.year > 2000 = F, t.year > 1980 = T}:
     {('The Shawshank Redemption', 1994, 3), ('Pulp Fiction', 1994, 4),
       ('Beetlejuice', 1988, 6)}
```

Example 2. Post-filter `title(title, year, id)`

```
{mi_idx.score > '8.0' = T}
     {('9.0', 1), ('9.3', 3), ('8.9', 4), ('9.2', 5)}
{mi_idx.score > '8.0' = F, mi_idx.score > '7.0' = T}
     {('7.5', 6), ('7.9', 7)}
```

Example 3. `movie_info_idx(score, movie_id)`

```
{t.year > 2000 AND mi_idx.score > '7.0' = T}:
     {('The Dark Knight', 2008, '9.0', 1), ('Avatar', 2009, '7.9', 7)}
{t.year > 2000 AND mi_idx.score > '7.0' = F, t.year > 1980 AND mi_idx.score > '8.0' = T}:
     {('The Shawshank Redemption', 1994, '9.3', 3), ('Pulp Fiction', 1994, '8.9', 4)}
```

Example 4. Joined `(title, year, score, id)`

the following non-bold lines. Looking at Example 1, we see that the tagged relation is made up of 7 tuples split into 2 relational slices; the two relational slices divide the tuples based on whether they were produced after 2000 or not. Example 4 features an example of an assignment to a *complex* predicate expression, which we define as a predicate expression containing either an AND or an OR. Note that in all examples, no tuple satisfies more than a single set of assignments present in a tagged relation, satisfying mutual exclusivity.

2.2 Filter

Given the previous setup, we can now describe how filter operators function in tagged execution. In traditional query execution, a filter operator applies a predicate expression to an input relation and outputs the subset of tuples which evaluate to true for that predicate expression. In tagged execution, each filter operator is given a *tag map* specifying which relational slices of the input tagged relation it should evaluate the predicate expression on and what to do with the results. An entry in the tag map has the following signature:

$$\langle \textit{in-tag} \rangle \rightarrow \{T : \textit{Optional} \langle \textit{pos-tag} \rangle,\ F : \textit{Optional} \langle \textit{neg-tag} \rangle\}$$

The predicate expression is applied to all relational slices which have a tag that matches $\langle \textit{in-tag} \rangle$. If the optional $\langle \textit{pos-tag} \rangle$ is specified, then the tuples which evaluate to true are stored as a relational slice in the output tagged relation with the tag $\langle \textit{pos-tag} \rangle$. Similarly, if $\langle \textit{neg-tag} \rangle$ is specified, the tuples which evaluate to false form a relational slice with the tag $\langle \textit{neg-tag} \rangle$. Both $\langle \textit{pos-tag} \rangle$ and $\langle \textit{neg-tag} \rangle$ may be specified in a single entry. All relational slices which do not have any matching entries in the tag map are passed untouched to the output tagged relation. Note that same tag may appear as an output in multiple entries. In this case, relational slices which share the same output tag are merged together in the output tagged relation.

As an example, consider the filter operator with the predicate expression `t.year > 1980` being given a tag map with one entry:

$$\texttt{\{t.year > 2000 = F\}} \rightarrow \{T : \texttt{\{t.year > 2000 = F, t.year > 1980 = T\}}\}$$

When the tagged relation from Example 1 is passed in as input, only the second relational slice has a matching entry. Thus, the predicate expression is only evaluated on the tuples from that relational slice. The tuples which evaluate to true come together to form a new relational slice with the tag **`{t.year > 2000 = F, t.year > 1980 = T}`**. Meanwhile, the $\langle \textit{neg-tag} \rangle$ is missing from the tag entry, so tuples which evaluate to false are removed and not included in the output tagged relation. The first relational slice does not match any entries in the tag map and is passed untouched to the output tagged relation. The resulting tagged relation is shown in Example 2.

The planner determines the tag map for each filter operator. During runtime, the execution engine simply follows the instructions encoded in the tag map. As such, the degree of sophistication of the planner can have a large impact on performance, and a naive planner, such as one that produces a tag map entry for every relational slice in the input tagged relation and outputs all resulting true/false relational slices, can lead to poor runtimes. Thus, we show in Section 3 how our planners build effective tag maps for efficient execution. In the case of this example, the planner was intelligent enough to realize that titles produced after 2000 are also produced after 1980 and reduces the filter operator's work by omitting the tag map entry for **`{t.year > 2000 = T}`**.

2.3 Join

Similar to filter operators, join operators in tagged execution are given tag maps which determine how to combine their inputs during runtime. Assuming a join operator has an input "left" tagged relation and an input "right" tagged relation, an entry in the tag map has the following signature:

$$(\langle left\text{-}tag \rangle, \langle right\text{-}tag \rangle) \rightarrow \langle out\text{-}tag \rangle$$

For every pairing between a left relational slice and a right relational slice, if it has a matching $(\langle left\text{-}tag \rangle, \langle right\text{-}tag \rangle)$ entry, the tuples in those relational slices are joined to create an output relational slice with the tag $\langle out\text{-}tag \rangle$. Similar to filters, output relational slices which share the same tag are merged together in the output tagged relation. However, unlike filter operators, relational slices without a matching tag map entry are discarded and not included in the output tagged relation. Note the same join constraint is used for every pairing of relational slices.

As an example, consider the join between the tagged relations in Example 2 (left) and Example 3 (right) using the join constraint `title.id = movie_info_idx.movie_id` given a tag map with the following entries:

```
({t.year > 2000 = T}, {mi_idx.score > '8.0' = T})
            → {t.year > 2000 AND mi_idx.score > '7.0' = T}
({t.year > 2000 = T}, {mi_idx.score > '8.0' = F, mi_idx.score > '7.0' = T})
            → {t.year > 2000 AND mi_idx.score > '7.0' = T}
({t.year > 2000 = F, t.year > 1980 = T}, {mi_idx.score > '8.0' = T})
            → {t.year > 2000 AND mi_idx.score > '7.0' = F,
               t.year > 1980 AND mi_idx.score > '8.0' = T}
```

The first and second entries join the first relational slice from Example 2 with the first and second relational slices respectively from Example 3 to generate the relational slice with tag **`{t.year > 2000 AND mi_idx.score > '7.0' = T}`**. The third entry joins the second relational slice from Example 2 with the first relational slice from Example 3 to produce the relational slice with movies produced after 1980 with a score above 8.0 not included in the first output relational slice. Just as with filter operators, the planner determines the tag map, and intelligent planners can choose to

selectively join only the tuples that satisfy the overall predicate expression and avoid performing costly joins between relational slices whose results would be thrown away later in the pipeline. In this example, the entry (**`{t.year > 2000 = F, t.year > 1980 = T}`**, **`{mi_idx.score > '8.0' = F, mi_idx.score > '7.0' = T}`**) is omitted from the tag map because the planner recognizes that movies produced between 1981 and 2000 with a score between 7.1 and 8.0 do not satisfy the overall predicate expression.

2.4 Projection

A projection operator in tagged execution offers one final chance to filter based on tags before actual projection. The operator is given a set of tags by the planner, and only the tuples in relational slices which have a matching tag are projected. In the case of Example 4, all relational slices are required as results to Query 1, so the planner would include both tags for the projection operator.

2.5 Implementation

The exact implementation of tagged relations and the previously discussed operators can have a large impact on the overall runtime performance. Here, we discuss the details of how tagged execution is implemented in our system Basilisk.

2.5.1 Tagged Relation. Basilisk is a column-oriented system, so intermediate representations of relations contain tuples of indices rather than tuples of actual values. Each n-tuple is the result of joining n tuples and contains indices into the n tables. The actual tuple of values can be reconstructed by doing an index-based lookup into each table when needed. Given a relation of such tuples, tagged relations are constructed by creating an accompanying hash table of bitmaps. Tags serve as keys to the hash table, and each bitmap specifies which tuples belong to which relational slice. Although an alternative implementation of separating out each relational slice into its own relation was considered, we found this version to be less performant because it required moving around tuples as opposed to manipulating bits in a bitmap.

2.5.2 Filter. There are two important implementation details for filter operators. First, when evaluating the predicate expression, Basilisk takes the union across all bitmaps which have matching tags, and the predicate expression is evaluated once on the tuples specified by the combined bitmap. This results in fewer I/O calls to read the underlying data values than evaluating the predicate expression separately for each relational slice. Second, filter operators do not actually modify the underlying, non-tagged relation. Instead, only the hash table of bitmaps is updated to mirror the output tags and corresponding relational slices. Even tuples which no longer belong to any relational slice remain in the relation because removing a tuple from the middle of a relation would require expensive modifications to every single bitmap.

2.5.3 Join. Basilisk uses a hash join for all its joins. However, rather than building a separate hash table for each relational slice, Basilisk builds one giant hash table for all the relational slices. The values of the hash table contain enough supplementary details to determine which relational slices the key belongs to. This significantly improves runtime performance because relational slices can share join key values and only one hash table needs to be allocated.

3 TAG MANAGEMENT

The previous section described the mechanism by which tagged execution query operators can use tags during runtime to process only a subset of the input tagged relations. We now turn our attention to the problem of tag management; for each query, how should the planner decide which tags to use, and how should it build each tag map. This is important, since the tag maps ultimately

dictate how much work is done for each query, and a naive strategy to tag management can lead to an exponential number of tags, causing the overhead of tagged execution to outweigh the benefits. To demonstrate, we first describe this naive strategy in Section 3.1, and we show that while it is capable of accomplishing disjunctive predicate pushdown, its weaknesses make it inferior to traditional execution strategies. We then introduce the concept of *generalizing* tags in Section 3.2 and show that by using these generalized tags instead of the tags in the naive strategy, we can reduce the total number of tags in the system and avoid the exponential blowup in many cases. In Section 3.3, we detail how to build efficient tag maps using the generalized tags to reduce the amount of unnecessary work done by the system. Finally, we discuss how to extend our work to accommodate NULLs and three-valued logic in Section 3.4.

3.1 Naive Strategy

In the most naive strategy to tag management, each tag contains true/false assignments to only base predicates (as opposed to the arbitrarily complex predicate expressions described in Section 2.1). Tags like **`{t.year > 2000 = T, mi_idx.score > '7.0' = F}`** are valid, while **`{t.year > 2000 AND mi_idx.score > '7.0' = F}`** are not because it contains an AND. The tag map for each operator is built as follows. First, base tagged relations, created directly from the base tables, contain only one relational slice with the "empty" tag which contains no assignments (i.e, **`{}`**). For filter operators, create a tag map entry for each input tag and output both the "true" and "false" results. More formally, if the filter operator's associated predicate is P, for each input tag I, create the tag map entry:

$$I \rightarrow \{T : I \cup \{P = T\},\ F : I \cup \{P = F\}\}$$

For join operators, take the full Cartesian product of the input left and right tags. In other words, for each pairing of left and right input tags (L, R), generate the following tag map entry:

$$(L, R) \rightarrow L \cup R$$

Finally, the projection operator's set of allowed tags is the set of tags whose assignments satisfy the overall predicate expression.

This naive strategy is capable of performing disjunctive predicate pushdown. Since the predicate evaluation results are stored in tags, filter operators can be pushed down to the base table, and the projection operator simply selects all tags which satisfy the query's overall predicate expression. However, the naive strategy suffers from two large weaknesses. (1) Filter operators output both "true" and "false " each time, so they do not actually filter any tuples. Since the join operators' tag maps contain the full Cartesian product of input tags, this results in joins being performed on the full base tables. (2) In addition, each filter operator outputs two new tags for each input tag, multiplying the number of tags by two. Thus, after n filter operators, there will be 2^n output tags, and this exponential blowup can potentially cause the overhead to outweigh the benefits.

3.2 Tag Generalization

To address the weaknesses of the naive strategy, we introduce the idea of generalizing tags. Tag generalization is a technique which generalizes the assignments in a tag based on the structure present in the query's predicate expression and Boolean implication. A generalized tag can be used in place of an ungeneralized tag, and multiple ungeneralized tags may generalize to the same generalized tag. Thus, replacing all the tags in the naive strategy with generalized tags reduces the total number of tags in the system and avoids the exponential blowup in many cases. An additional benefit of tag generalization is that it clearly exposes (as early as possible) which tags will not be part of the final output. The planner can then use this information to build efficient tag maps which discard tuples that do not satisfy the overall predicate expression as early as possible (Section 3.3).

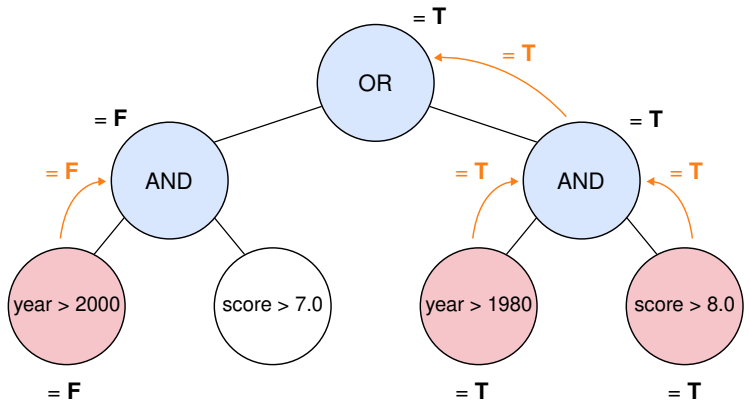


Fig. 2. Propagation with predicate tree from Query 1.

In tag generalization, the query's predicate expression is represented as a *predicate tree*, in which each node represents a subexpression. Leaf nodes represent the base predicates, and intermediate AND/OR/NOT nodes apply their operations to their children to form their subexpressions[3]. The root node refers to the entire predicate expression. Figure 2 shows the predicate tree for Query 1.

Tag generalization works by propagating a tag's assignments upwards in the predicate tree. As an assignment travels upwards, it becomes more general and encompasses more possible tags. For example, the assignment `t.year > 2000 = F` can be generalized into the assignment `(t.year > 2000 AND mi_idx.score > '7.0') = F` because the former implies the latter according to Query 1's predicate tree. The generalized tag **`{(t.year > 2000 AND mi_idx.score > '7.0') = F}`** can then be used in place of not only **`{t.year > 2000 = F}`**, but also tags such as **`{mi_idx.score > '7.0' = F}`** and **`{t.year > 2000 = T, mi_idx.score > '7.0' = F}`** because these tags also imply the generalized tag.

GeneralizeTag from Algorithm 1 (next page) outlines the general algorithm for generalizing tags. The input to GeneralizeTag is a tag which maps predicate expressions to true/false values; each key-value pair constitutes an assignment. At its core, GeneralizeTag is a fringe-based algorithm which propagates a predicate expression's assignment to its parent if the parent's assignment can be implied by its children. The fringe is initialized with the set of all predicate expressions with an assignment in the input tag (i.e., "keys(tag)"). The function canPropagate checks for the condition of Boolean implication and returns true if one of five conditions is satisfied: (a) If the parent is a NOT node. (b) If the predicate expression's assignment value is true and the parent is an OR node. (c) If the predicate expression's assignment value is false and the parent is an AND node. (d) If the parent is an OR node and all its children have false assignments. (e) If the parent is AND node and all its children have true assignments. The actual propagation is done with doPropagate, and unless the parent is a NOT node, the predicate expression's assignment value is assigned directly to the parent. If the propagation is successful, the parent is added to the fringe, and the loop continues. Note that the same predicate expression may appear multiple times in the predicate tree, so the "parents" function returns the parent for each instance. Once all assignments have been propagated as much as possible, topmostAssignments is called on the resulting tag to collect only the topmost assignments. Assignments to predicate expressions, which are descendants of other

[3]The predicate tree is also normalized such that an intermediate node cannot be of the same type as their parent (i.e., an AND node's parent cannot also be an AND node)

Algorithm 1 GeneralizeTag

Input: Tag "tag"

```
fringe ← keys(tag)
while fringe.isNotEmpty() do
    pred ← fringe.pop()
    for parent ∈ parents(pred) do
        if canPropagate(pred, parent, tag) then
            doPropagate(pred, parent, tag)
            fringe.push(parent)
return topmostAssignments(root, tag)

function canPropagate(pred, parent, tag)
    return isNot(parent) or
      (isOr(parent) and tag[pred] = T) or
      (isAnd(parent) and tag[pred] = F) or
      (isOr(parent) and ∀c ∈ children(parent), tag[c] = F) or
      (isAnd(parent) and ∀c ∈ children(parent), tag[c] = T)

function doPropagate(pred, parent, tag)
    if isNot(parent) then
        tag[parent] ← ¬tag[pred]
    else
        tag[parent] ← tag[pred]

function topmostAssignments(node, tag)
    if tag = {} then
        return {}
    else if node ∈ keys(tag) then
        return {node : tag[node]}
    else
        return ⋃_{c∈children(node)} topmostAssignments(c, tag)
```

predicate expressions with assignments, are all are discarded. Note that GeneralizeTag runs in $O(n)$ time, in which n is the number of predicates.

Figure 2 shows the predicate tree for Query 1 undergoing this process for the tag **`{t.year > 2000 = F, t.year > 1980 = T, mi_idx.score > '8.0' = T}`**. The red-colored leaf nodes mark the nodes with the initial assignments, the orange arrows show the propagation, and the blue-colored intermediate nodes mark the nodes which get assignments due to propagation. After topmostAssignments is called, only the assignment at the root node remains, and the output of GeneralizeTag is **`{(t.year > 2000 AND mi_idx.score > '7.0') OR (t.year > 1980 = T AND mi_idx.score > '8.0') = T}`**.

Using the predicate tree to generalize assignments also has the benefit of making it clear which predicate expressions still need to be applied to which tags. In the previous example, the tag after propagation **`{(t.year > 2000 AND mi_idx.score > '7.0') OR (t.year > 1980 = T AND mi_idx.score > '8.0') = T}`** has a true assignment to the overall expression, signifying that every tuple with this tag should be part of the final output. Thus, no additional filters need to be run on this tag. Similarly, if the overall predicate expression has a false assignment, it signifies that every tuple

with the tag does not match the criteria for the query, and these tuples can safely be removed from the pipeline.

Duplicates. As mentioned, the same predicate expression may appear multiple times in the overall predicate expression. In fact, conversations with real users of disjunctive workloads have led us to believe that this is actually a common occurrence for disjunctive queries. Thus, careful consideration has gone into the development of GeneralizeTag and topmostAssignments to handle this case. Specifically, GeneralizeTag allows different levels of propagation for different instances of the same predicate expression, and topmostAssignments only removes a predicate expression's assignment if every instance has an ancestor with an assignment. It is this treatment of predicate expressions which allows tagged execution to evaluate each predicate exactly once.

Limitations. Although generalizing tags reduces the tag space in most cases, the number of tags produced can still be exponential in the worst case. Consider the predicate expression with the form $(X_1 \vee Y_1) \wedge (X_2 \vee Y_2) \wedge ... \wedge (X_n \vee Y_n)$, in which all X_i and Y_i are predicates. If the filter operators are ordered and applied in $X_1, X_2, ..., X_n, Y_1, Y_2, ..., Y_n$ order, then 2^n tags are still required to keep track of the unique relational slices. However, such degenerate predicate expressions are not so common in practice, and the planner can interleave the Y_i predicates (i.e., use a different plan) to reduce the number of tags.

3.3 Building Tag Maps

We now describe how the planner should build the tag maps for each operator in a given query plan. Tag maps should minimize the amount of work performed by operators during runtime while still ensuring correctness of the query plan. To accomplish this, our planners follow two main precepts when building tag maps: (1) Avoid generating tags which do not get used by the rest of the pipeline. (2) Do not apply filters on tags if it does not help refine the selection process. Fortunately, tag generalization make it easy to recognize both cases.

For (1), we can identify such tags if they include a false assignment to the root node of the predicate tree. Tuples with this tag are guaranteed to not satisfy the overall predicate expression and can be discarded without further consideration. On the other hand, if an assignment to the root node does not emerge after generalizing a tag, it signifies that a portion of the tuples associated with that tag may still satisfy the overall predicate expression, and more filter operators must be applied to refine the associated relational slice. For example, in Figure 1, the filter operator with the predicate expression `t.year > 2000` is applied before the filter operator with the predicate expression `t.year > 1980`. The first filter operator results in the tags **`{t.year > 2000 = T}`** and **`{t.year > 2000 = F}`**, and after tag generalization, **`{t.year > 2000 = F}`** becomes **`{t.year > 2000 AND mi_idx.score > '7.0' = F}`**. However, the relational slice with the false tag may still contain tuples that satisfy the overall predicate expression, namely the movies produced after 1980 with a score greater than 8.0. Thus, the relational slice must be kept. However, after applying the second filter operator to this relational slice, movies produced before 1980 will generate the tag **`{t.year > 2000 AND mi_idx.score > '7.0' = F, t.year > 1980 = F}`**, and this generalizes to **`{(t.year > 2000 AND mi_idx.score > '7.0') OR (t.year > 1980 AND mi_idx.score > '8.0') = F}`**. At this point, we can be sure the relational slice with this tag does not contain any tuples which satisfy the overall predicate expression, since movies produced before 1980 do not satisfy the overall criteria. Thus, for the second filter operator, the planner should omit the negative output tag ($\langle neg\text{-}tag \rangle$) from the tag map entry.

For (2), let us assume we apply the filter operators with `t.year > 2000` and `mi_idx.score > '7.0'` in that order after joining the tables. In this case, applying the predicate expression `mi_idx.score > '7.0'` to the relational slice with the tag **`{t.year > 2000 = F}`** is pointless. The tuples with this tag are already guaranteed to not satisfy the first disjunctive clause in Query 1,

so dividing this set of tuples into those that satisfy/do not satisfy `mi_idx.score > '7.0'` does nothing to help the tuple selection process. Precept (2) states that such cases should be avoided, and generalized assignments make it trivial to identify such cases. The tag **`{t.year > 2000 = F}`** after generalization becomes **`{t.year > 2000 AND mi_idx.score > '7.0' = F}`**, and this predicate expression is an ancestor (in the predicate tree) of the predicate expression we were trying to apply (`mi_idx.score > '7.0'`). Thus, both positive and negative outcomes of `mi_idx.score > '7.0'` (i.e., **`{t.year > 2000 AND mi_idx.score > '7.0' = F, mi_idx.score > '7.0' = T}`** and **`{t.year > 2000 AND mi_idx.score > '7.0' = F, mi_idx.score > '7.0' = F}`**) reduce to the same tag **`{t.year > 2000 AND mi_idx.score > '7.0' = F}`** after generalization, signifying that the application of `mi_idx.score > '7.0'` to this tag was not very helpful in refining the tuples according to overall predicate expression.

With these precepts in mind, we now describe the exact process our planners follow for building tag maps. The overall construction is similar to the naive strategy, except it includes tag generalization and the two precepts. First, base tagged relations, created directly from the base tables, contain only one relational slice with the "empty" tag which contains no assignments (i.e, **`{}`**).

Filter. For filter operators, if the associated predicate expression is P, for each input tag I:

(1) If each instance of P in the predicate tree has an ancestor with an assignment in I, do nothing (Precept (2)).
(2) Otherwise, run GeneralizeTag on both positive and negative output tags:

$$P \leftarrow \textsc{GeneralizeTag}(I \cup \{P = T\})$$
$$N \leftarrow \textsc{GeneralizeTag}(I \cup \{P = F\})$$

And create a tag map entry for I:

$$I \rightarrow \{T : P,\ F : N\}$$

If either P or N include a false assignment to the root node, remove it as an output tag from the entry (Precept (1)).

Join. For join operators, take the full Cartesian product of the input left and right tags. For each pairing of left and right input tags (L, R), generate the output tag by generalizing the union:

$$O \leftarrow \textsc{GeneralizeTag}(L \cup R)$$

If O does not have a false assignment to the root node (Precept (1)), create an entry in the tag map:

$$(L, R) \rightarrow O$$

Projection. For projections operators, restrict the set of allowed tags to only the tag with a true assignment to the root node.

3.4 Extension to Three-Valued Logic

Standard SQL allows for NULL values, and evaluating a predicate on a NULL value often results in a non-true/false, ternary value called "unknown". Fortunately, our framework extends very naturally to this three-valued logic, and only four changes need to be made:

(1) Instead of restricting a tag's assignments to either be true/false, a tag's assignments may now be one of true/false/unknown.
(2) For filter operator tag map entries, include an optional output ⟨*unknown-tag*⟩ for unknown results.
(3) The functions canPropagate and doPropagate in Algorithm 1 must be updated to handle unknown values. For canPropagate, the only difference is for conditions (d) and (e), which must now be changed to: (d) If the parent is an OR node and all its children have false or

unknown assignments. (e) If the parent is AND node and all its children have true or unknown assignments. On the other hand, doPropagate must now update the parent's assignment value based on the three-valued logic from the SQL standard [29] (e.g., false OR unknown → unknown).

(4) Finally, applications of Precept (1) must be changed so that tag map entries do not contain output tags with either a false or unknown assignment to the root node.

4 PLANNING

Now that we understand how to execute queries under tagged execution, and how to build efficient tag maps which effectively prune tuples from disjunctive queries early, we describe how we can generate plans that take advantage of these early filtering capabilities. We begin by presenting cost models for tagged execution (Section 4.1) and then give the details for several planners (Section 4.2). Note that it is not the goal of this work to produce the most advanced, optimal planner for tagged execution. Rather, we present a few simple planners which highlight the advantages of tagged execution and demonstrate that even these simple planners can obtain substantial performance compared to existing methods.

4.1 Cost Models

Although tagged execution introduces a new model for query execution, internally, tagged operators employ the same relational operators from traditional query execution. The only difference is that the input changes from whole relation(s) to individual relational slice(s). As such, our suggested cost models mirror existing models quite closely, differing mainly in that our cost models are summations of the costs of individual relational slices.

For filter operators, the cost model is:

$$C_{filter} = \alpha \sum_{I \in M} F_P \, |R[I]|$$

Here, C_{filter} measures the cost of applying the filter operator with predicate expression P on input tagged relation R. The expression $I \in M$ iterates over each input tag in the given tag map, and F_P is a certain constant cost factor specific to P. The expression $R[I]$ selects the relational slice associated with tag I in R, and $|R[I]|$ measures the cardinality of the selected relational slice. Finally, α is a constant cost factor used to calibrate filter costs with respect to join costs. In short, the total cost is the summation of the costs from applying predicate expression P to every relational slice whose tag appears as an input in the given tag map.

For join operators, the cost model is:

$$C_{join} = C_{hash\_build} + C_{hash\_lookup} + C_{index\_build}$$

$$C_{hash\_build} = F_{hash\_lookup} \left|R'_{left}\right| + F_{hash\_build} \cdot \text{unique}(R'_{left})$$

$$C_{hash\_lookup} = F_{hash\_lookup} \left|R'_{right}\right|$$

$$C_{index\_build} = F_{index\_build} \left|R'_{left} \bowtie R'_{right}\right|$$

As mentioned, all join operators in our system are implemented as hash joins, so the cost of a join can be split into three components: (1) C_{hash_build} – the cost of building a hash map from the left input tagged relation[4], (2) C_{hash_lookup} – the cost of performing the hash lookups from the right

[4]Although this cost estimate assumes the hash map will be built from the left side, our system actually makes an estimate from both sides and chooses the cheaper side.

input tagged relation, and (3) C_{index_build} – the cost of building the index for the output joined tagged relation. Building the hash map requires performing a hash lookup for each element and creating entries for each unique element. Accordingly, F_{hash_lookup} is some cost factor associated with hash lookups, $|R'_{left}|$ measures the cardinality of R'_{left}, F_{hash_build} is some cost factor associated with creating hash map entries, and unique(R'_{left}) counts the number of unique elements in R'_{left}. Note that R'_{left} is not the left input tagged relation; rather, R'_{left} is the union of relational slices in the left input tagged relation whose tags have at least one matching entry in the given tag map; R'_{right} from C_{hash_lookup} is defined similarly. Finally, the cost of building the index for the output joined tagged relation is the product of some cost factor F_{index_build} and its cardinality.

Note that the cost models for both filter and join operators require cardinality estimates for tagged relations/relational slices. In general, the cardinality estimate for a tagged relation is given as the sum of its relational slices's cardinality estimates. For filters, we measure and use the selectivities of predicates along with the independence assumption. For joins, we use PostgreSQL's cardinality estimations of joins [27].

4.2 Planners

Here, we present several planners for tagged execution. Each planner optimizes for a different situation, and in our system, we use the TCombined planner, which estimates the cost of the plan produced by each of the following planners and selects the cheapest one.

TPushdown. This planner creates a filter operator for each predicate in the predicate expression and pushes all filter operators down to the base table level. All joins are performed after the filter operators and are ordered greedily; whichever join would produce the smallest cardinality tagged relation is performed next (this is actually the join ordering used for all our planners). After pushdown, if there are multiple filter operators for a single table, they are sorted in *benefiting* order. The benefit score of a filter operator is calculated with respect to a set of filter operators, and it estimates the benefit of applying that filter operator first before the set of filter operators. The score is used here and in other planners to avoid materializing every alternative plan and serves as an effective proxy for plan cost. The exact method to calculate the benefit score can be found in our technical report [24]. TPushdown is the "naive" planner which simply pushes down all all filter operators down to the base table level. It serves as a good baseline planner, since for many queries, the joined relation is much larger than the base table relations, and pushing filter operators down to the base table level prunes as many tuples as early as possible. Figure 1 serves as an example query plan that TPushdown might produce.

Algorithm 2 TPullup

```
(best_plan, best_cost) ← TPushdown()
for filter ∈ filters(plan) do
    new_plan ← best_plan
    while can_pullup(new_plan, filter) do
        (new_plan, new_cost) ← pullup_node(new_plan, filter)
        if new_cost < best_cost then
            (best_plan, best_cost) ← (new_plan, new_cost)
return best_plan
```

TPullup. The pseudocode for this planner is presented in Algorithm 2. This planner uses the plan produced by TPushdown, in which all filter operators are pushed down, as the base plan. Then, for each filter operator (in reverse benefiting order), the planner considers pulling up [14]

the operator by one node in the query plan, and if the resulting plan is cheaper, that plan is used as the base plan from then on. In the end, the planner returns the cheapest plan it finds. TPullup builds on TPushdown for situations in which certain predicate subexpressions are very selective and can cause the joined result to be much smaller than the base tables. In these cases, it may be cheaper to delay the application of other more expensive subexpressions, so TPullup pulls up each filter node to check. For example, consider the predicate expression: `(mi_idx.score = '9.2' OR mi_idx.score = '9.3') AND t.title ILIKE '%godfather%'`. There are only two movies in the IMDB dataset with a score above 9.0, so applying the score predicates to the `movie_info_idx` table and joining it with the `title` table will lead to a joined relation of two tuples. As such applying the expensive regex pattern matching predicate on this joined result may result in a cheaper plan than applying it directly to the entire `title` table. TPullup can check for situations such as this and output the following plan:

```
Filter(t.title ILIKE '%godfather%')
 └─Join(t.id = mi_idx.movie_id)
    ├─Table(title as t)
    └─Filter(mi_idx.score = '9.2')
       └─Filter(mi_idx.score = '9.3')
          └─Table(movie_info_idx as mi_idx)
```

TIterPush. This planner operates in the "opposite" direction of TPullup. The base plan performs all joins first, then all filters are applied in benefiting order. For each filter operator (in benefiting order), the planner considers pushing down the operator to the base table level. If the resulting plan is cheaper, that plan is used as the base plan from then on. In the end, the planner returns the cheapest plan it finds. One weakness of TPullup is that it only considers moving one filter operator at a time. Sometimes, there is no benefit to moving a single operator; only when multiple filter operators are moved can a change in the plan cost be observed. Thus, TPullup does not always find the optimal plan, and TIterPush serves as the opposite extreme, in which the default for all filter operators is to be performed after all the joins, and filter operators are only pushed down if it leads to a cheaper plan. To take an example, let us consider the predicate expression: `mi_idx.score > '9.0' AND (t.title ILIKE '%godfather%' OR t.title ILIKE '%lord%')`. Once again, we would like to perform the expensive regex pattern matching predicates on the joined result, since `mi_idx.score > '9.0'` is so selective. Consider the plan in which only `t.title ILIKE '%godfather%'` is pulled up:

```
Filter(t.title ILIKE '%godfather%')
 └─Join(t.id = mi_idx.movie_id)
    ├─Filter(t.title ILIKE '%lord%')  ← (1)
    │  └─Table(title as t)
    └─Filter(mi_idx.score > '9.0')
       └─Table(movie_info_idx as mi_idx)
```

This plan is not any cheaper compared to the plan in which all filters are pushed down. This is because the tagged relation at (1) after the filter must include both **`{t.title ILIKE '%lord%' = T}`** and **`{t.title ILIKE '%lord%' = F}`**, so the join operator ends up joining to the entire `title` table. In comparison, the base plan applies both parts of the disjunct to `title`, and only the tuples with the tag **`{t.title ILIKE '%godfather%' OR t.title ILIKE '%lord%' = T}`** are joined. TPullup is incapable of recognizing that pulling up both regex predicates would lead to a more optimal plan. Thus to provide another perspective for these situations, TIterPush starts with all filters pulled up,

and in this case pushes down only the `mi_idx.score > '9.0'` predicate to arrive at the following plan:

```
Filter(t.title ILIKE '%godfather%')
 └─Filter(t.title ILIKE '%lord%')
    └─Join(t.id = mi_idx.movie_id)
       ├─Table(title as t)
       └─Filter(mi_idx.score > '9.0')
          └─Table(movie_info_idx as mi_idx)
```

TPushConj. This planner mimics what existing planners might do for disjunctive predicates if the root node of the predicate tree is an AND node. Children of the root node which are individual predicates or predicate expressions whose descendent predicates all apply to the same table are pushed down. The remaining children are performed in increasing order of selectivity after all the joins. In short, the planner pushes down the conjunctive filter operators it can, and resolves the remaining after the joins. This planner mostly serves as a comparison point to traditional query execution. Although it would arrive at the same plan as TPushdown for the previous example (in TIterPush), it would not be able to perform any pushdown optimizations for simple CNFs of the form $(P_1 \vee P_2) \wedge (P_3 \vee P_4)$, if P_1 and P_2 refer to different tables and P_3 and P_4 refer to different tables.

Discussion. Note that in all these planners, we could reorder and push down/pull up predicates however we wanted, without having to worry about how it affects the evaluation of the query's predicate expression. This is one of the highlights of the tagged execution abstraction model. By encapsulating all the state required to evaluate the predicate expression into tags, tagged execution is able to disentangle the complexities of evaluating a disjunctive predicate expression from the query plan. As a result, planners for tagged execution can rearrange predicates freely, almost as if they are planning for a predicate expression with only conjunctions. Not only does this provide a cleaner interface, it also reduces the query plan space, allowing our planners to complete faster.

5 EVALUATION

For our evaluation, we wish to compare the runtime performance of the tagged execution model using our planners against the traditional execution model using existing planners. In addition, we wish to measure the "overhead" of executing queries under the tagged execution model and explore how different query parameters can affect the level of benefit that the tagged execution model has to offer. To this end, we executed queries from the Join Order Benchmark (JOB) [27] (Section 5.1) and a set of synthetic experiments varying different query parameters (Section 5.2) under both tagged and traditional execution models and measured their total runtimes. To represent traditional execution, we implemented the following two planners:

(1) BDisj is for predicate expressions with OR root nodes in the corresponding predicate trees (e.g., DNFs). The planner treats each *root clause* (i.e., child of the root node) as a separate query; each root clause is executed independently of the others, applying pushdown optimizations wherever possible, and a final union operator to combine all the results. This (or a variation) is the approach taken by many academic papers [36] [18] [30] [4], and experts recommend rewriting SQL queries to achieve this manually for systems which do not support this internally [1].
(2) BPushConj is for predicate expressions with AND root nodes (e.g., CNFs) and is the counterpart to TPushConj for traditional execution. Every root clause whose predicate descendants all refers to the same table are pushed down to that table, and the remaining root clauses are performed after all the joins. This (or a variation) is the approach taken by most real-world systems, such as PostgreSQL [35], Hyrise [11], and Vertica [26].

Note that similar to the tagged execution planners from Section 4, both BDisj and BPushConj order joins greedily. We compared these traditional execution planners against our combined tagged execution planner TCombined with its subplanners TPushdown, TPullup, TIterPush, and TPushConj.

System. All experiments were conducted on our system Basilisk[5], a column-oriented database system capable of performing both traditional and tagged query execution. Basilisk is coded in ~12.6k lines of Rust, and data is stored on disk. When the data for a relational slice is needed, Basilisk consults the corresponding bitmap, and reads are done using direct I/O calls with a LFU page cache sitting in the middle. For bitmaps with low selectivity (i.e., only a few values need to be read), only the relevant pages are read from disk. However, doing the same for bitmaps with high selectivity (i.e., many values need to be read) can lead to substantial penalties from the random I/O, so for all bitmaps with a selectivity above a certain threshold, Basilisk instead reads the entire column sequentially, and values are selected in memory. For the experiments, we ran Basilisk on a server running Arch Linux with 40 Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz processors, 128GB of memory, and a SSD with 6Gbps of I/O throughput.

5.1 Join Order Benchmark

It should be noted that there is a distinct lack of publicly available disjunctive query workloads. Although real workloads containing disjunctions have been reported [8] [19], their proprietary nature prevents us from accessing them. Luckily, JOB provides an avenue to concoct some realistic disjunctive queries. Vanilla JOB does include a few queries with disjunctions, but these disjunctions never span more than a single table and are not fit for our workload by themselves. JOB sorts its queries into 33 distinct query groups. Each query group follows some sort of theme, and more importantly, all queries within a group operate on the same set of tables and use the same join conditions, differing only in the filtering predicate expressions. Thus, the queries in each query group can be combined together by taking the disjunction of their predicate expressions. For example, query group 20 is about superheroes; query 20a searches for superhero movies produced after 1950 with a character named "Iron Man", and query 20c searches for superhero movies produced after 2000 with just any character with the word "Man" in their name[6]. Combining queries 20a and 20c would give us one query which searches for superhero movies either produced after 1950 with a character named "Iron Man" or produced after 2000 with any character with the word "Man" in their name. Doing this for every query group provides us with 33 complex and realistic disjunctive queries to use for evaluation.

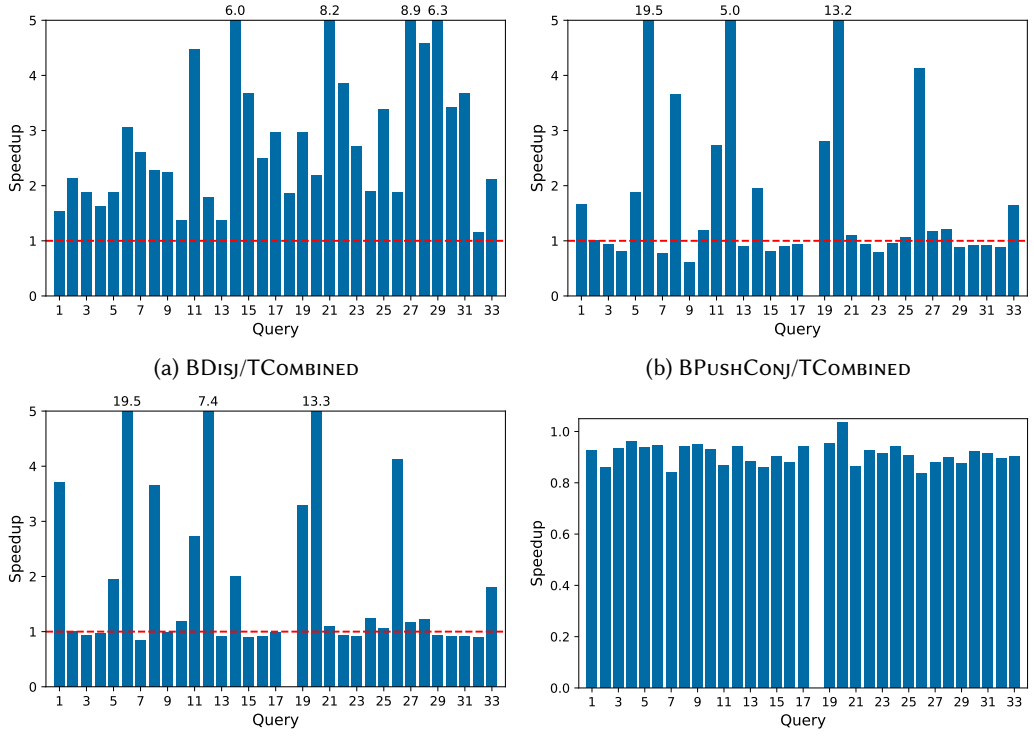
Figure 3 (next page) presents the results of our evaluation for JOB. We ran each query 5 times for each planner, and the figures use the average of those 5 times. We present only the total runtimes and do not present planning and executions times separately because in every case, planning time accounted for less than 0.1% of the total runtime.

Key Takeaways. We observed that tagged execution greatly outperformed traditional execution in many cases for JOB. Specifically, TCombined had an average 2.7× speedup over BDisj, and while the results varied more against BPushConj, there were still queries in which TCombined achieved a 19× speedup over BPushConj. At the same time, we observed that the tagged execution model incurred an average overhead of only 10% compared to traditional execution.

Figure 3a presents the speedups in total runtime achieved by TCombined over BDisj for each query. As shown clearly, TCombined achieved at least a 2× speedup for most queries and for queries

[5]https://github.com/alkim0/tagexec

[6]The actual queries are more complex, but this serves as a summary.



(a) BDisj/TCombined (b) BPushConj/TCombined

(c) BPushConj/TMin (d) BPushConj/TPushConj

Fig. 3. Speedups in total runtime from using tagged execution for JOB queries (> 1 means tagged execution is better).

27 and 21, achieved speedups of 9× and 8× respectively. Deeper analysis revealed a multitude of reasons for these results. First, the individual queries in each query group of JOB all shared common predicate subexpressions. However, since BDisj treats root clauses as completely independent of one another, the system ended up performing redundant work in evaluating the same predicate subexpression multiple times. In a similar vein, different root clauses often operated on the same set of tuples. Although Basilisk only materializes the indices for each tuple until projection (see Section 2.5), this still meant that the same tuple's indices needed be materialized multiple times in intermediate relations across different root clause executions. The final union operator that BDisj appends to remove duplicate tuples also incurred significant runtime for queries whose final output relation size was large. In contrast, TCombined only ever evaluated each predicate subexpression once, and each tuple is only ever materialized once as well. TCombined also does not require an additional union operator; the tag system is sufficient to track whether a tuple belongs to the final output or not. Finally, joins in TCombined often completed much quicker than the joins in BDisj, even for input (tagged) relations of similar sizes, displaying the importance of selective tag maps.

Since root clauses often shared common predicate subexpressions, we wondered what would happen if a traditional execution planner was smart enough to take advantage of that information. Thus, we searched for common predicate subexpressions that were children to every root clause in a query and pulled out those predicate subexpressions to create an equivalent predicate expression with an AND root node (e.g., a predicate expressions $(A \wedge B \wedge C) \vee (A \wedge B \wedge D)$ would be transformed

into $A \wedge B \wedge (C \vee D)$). We ran BPushConj and TCombined on the resulting queries, and Figure 3b shows the speedups achieved by TCombined over BPushConj. Note that BPushConj always ran out of memory before completing query 18, so the figure does not include a speedup for that query. For several queries, TCombined still achieved significant speedups. Specifically, for queries 6 and 20, TCombined achieved speedups of 19× and 13× respectively (greater than any speedup achieved over BDisj). For these queries, TCombined was able to push down all predicates, while BPushConj was only able to push down the predicates belonging to the common predicate subexpressions. This made a large difference because the unpushed predicates included several expensive predicates, such as regular expression matching, and the sizes of the relations after the joins were much larger than the sizes of the base tables. In addition, the tagged join operator continued to exhibit superior runtimes compared to the regular join operator even for similar sized input (tagged) relations. On the other hand, for many of the queries, the speedups ranged from 0.9 - 1.1, indicating similar performance between TCombined and BPushConj. The primary reason for this was due to the highly selective nature of the common predicate subexpressions. As mentioned, each query group in JOB follows some sort of theme, and the common predicate subexpressions often included highly selective predicates defining that theme. These highly selective predicates would filter most of the tuples early in the pipeline, preventing TCombined from really showcasing the benefits of tagged execution. Even worse, for cases such as query 9, the speedup dipped as low as 0.6×. However, this was mostly due to inaccurate cost models. Sometimes, TCombined would select the "cheapest" plan, only for a different tagged execution planner to outperform it. To account for this, we also measured speedup of the minimum runtime achieved by *any* tagged execution planner over BPushConj. Figure 3c shows the results. As can be seen, the minimum speedup becomes 0.8×, and the speedups of several queries jump even higher. This indicates that with a more realistic cost model, tagged execution could potentially achieve even greater speedups over traditional execution.

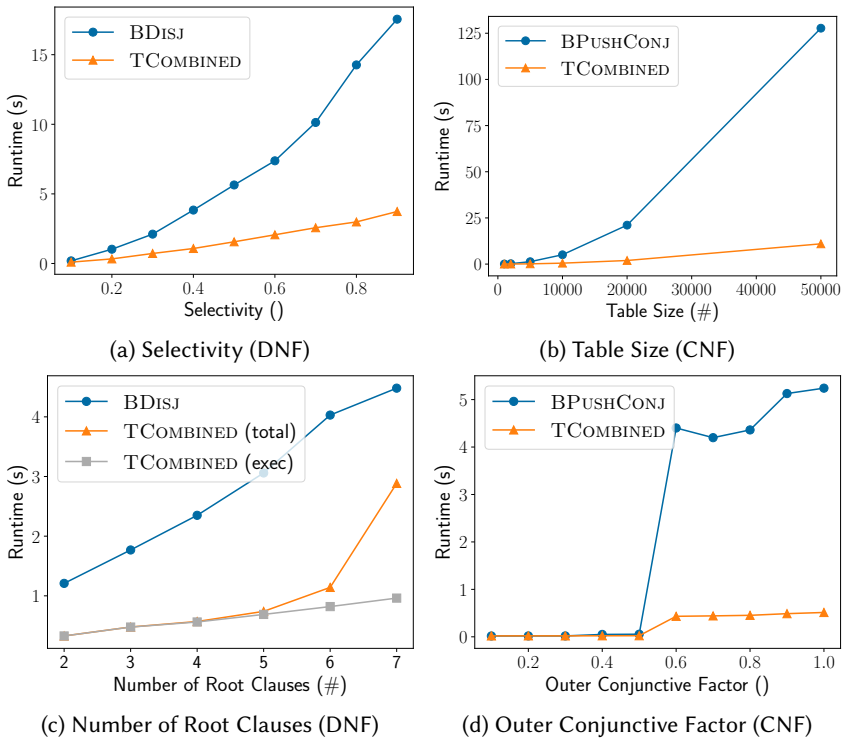
Finally, we wanted to measure the "overhead" of tagged execution with respect to traditional execution. That is, given a plan that can be run under both tagged and traditional execution models, how much slower is the tagged execution engine compared to the traditional execution engine in completing that plan. We accomplished this by looking at the speedups of TPushConj over BPushConj. The plan generated by these planners forces tagged execution to behave exactly like traditional execution. Filter operators for all pushed predicates do not include the $\langle$*neg-tag*$\rangle$ in their tag maps, and join operators take the full Cartesian product between their input left and right relational slices. Figure 3d shows the results. As can be seen, the average speedup is around 0.9×, suggesting a 10% overhead in using tagged execution over traditional execution.

5.2 Synthetic Experiments

We also evaluated the tagged execution model on a set of synthetic experiments varying different parameters. These experiments had two base queries; one in CNF and one in DNF. The DNF version of the query was:

```
SELECT * FROM T0 JOIN T1 ON T0.id = T1.fid
                 JOIN T2 ON T0.id = T2.fid
          WHERE (T1.A1 < 0.2 AND T2.A1 < 0.2)
             OR (T1.A2 < 0.2 AND T2.A2 < 0.2)
```

The CNF version of the query was the same, except the ANDs and ORs were swapped in the predicate expressions (i.e., `(T1.A1 < 0.2 OR T2.A1 < 0.2) AND (T1.A2 < 0.2 OR T2.A2 < 0.2)`). These are relatively simple queries, but because each root clause contains predicates referencing different tables, existing systems cannot optimize them very well. In fact, there is no way to optimize the CNF version using a traditional execution model. As for the underlying dataset,



(a) Selectivity (DNF) (b) Table Size (CNF)

(c) Number of Root Clauses (DNF) (d) Outer Conjunctive Factor (CNF)

Fig. 4. Synthetic experiments varying a number of variety of different parameters.

the tables `T0`, `T1`, and `T2` each had 10k records. Table `T0`'s `id` attribute was the primary key and had unique values ranging from 1 to 10,000. `T1` and `T2`'s `fid` attributes were the foreign keys, and their values were randomly generated using a Zipf distribution with a shape parameter value of 1.5. The attributes that appeared as part of predicates (e.g., `T1.A1` and `T2.A2`) had values ranging from 0 to 1, generated uniformly at random.

From these base queries and dataset, we varied a number of various parameters. For DNF queries, we ran BDisj and TCombined, while for CNF queries, we ran BPushConj and TCombined. Figure 4 shows some of the results. In every experimental configuration, each query was run 5 times for each planner, and the figures report the average of those 5 times. With the exception of the experiment varying the number of root clauses, planning time once again accounted for less than 0.1% of the total runtime, so only Figure 4c plots the total runtime and the execution runtime separately.

Key Takeaways. We observed that as the number of tuples handled by a query increased, tagged execution increasingly outperformed traditional execution. Specifically, TCombined achieved speedups of up to 12× over BDisj for DNF queries and speedups of up to 10× over BPushConj for CNF queries.

Selectivity. The first parameter we varied was the selectivity of the predicates, which we varied from 0.1 to 0.9. Figure 4a presents the results for DNF queries. As shown, the runtimes diverged drastically as the selectivity value grew, with TCombined achieving a speedup of 5× over BDisj when selectivity is 0.9. When selectivity was small, most tuples were filtered out early, so the difference between BDisj and TCombined was not as drastic. However, as the selectivity grew, the sizes of the intermediate and resulting relations also grew, and this had a greater impact on BDisj

than TCombined for three reasons. First, even though the joins had similar size inputs for both tagged and traditional execution, the selective tag maps in tagged execution reduced the amount of work performed by each join operator. Second, as selectivity grew, more duplicate tuples were materialized in intermediate relations across different root clauses for BDisj, while TCombined only materialized each tuple once. Third, the union operator for BDisj handled more tuples with increasing selectivity, resulting in more overhead. The plot for the CNF version of the query had a slightly different shape. Instead of the runtimes differing as a function of selectivity, there was instead a constant (large) difference between the runtimes for TCombined and BPushConj (TCombined was faster). Due to each root clause of the CNF referring to multiple tables, BPushConj could not push down any root clauses, so all filter operations were performed after all the joins. Thus, the time taken to perform the join operations (the largest factor) remained constant even as selectivity varied.

Table Size. Next, we varied the table sizes of `T0`, `T1`, and `T2` from 1k records up to 50k records. Figure 4b presents the results for CNF queries. The plot displays the same trends as Figure 4a, with TCombined achieving a speedup of 12× when the table size is 50k records. The reasons were similar as well. With larger table sizes, the sizes of intermediate and resulting relations also grew, and this affected BPushConj more than TCombined. Because BPushConj could not push any predicates down, it suffered directly from the quadratic growth in the join result. On the other hand, TCombined could execute all its filter operators on the base tables, which grew linearly, and the join operators in tagged execution benefited significantly from the use of tag maps. The plot for the DNF version of the query had the same shape, and the reasons were the same as those mentioned for the DNF version of the selectivity experiment.

Number of Root Clauses. We also varied the number of root clauses from 2 to 7. Each additional root clause referred to new attributes in `T1` and `T2` (e.g., `T1.A3` and `T2.A3`). Figure 4c presents the results for DNF queries. Note that this was the only experiment in which the planning time for TCombined started to have a noticeable impact on the overall runtime. As such, Figure 4c plots TCombined (total) for the total runtime and TCombined (exec) for only the execution runtime (without planning). First, note that the difference in runtime between BDisj and TCombined (exec) increases with more root clauses; at 7 root clauses, TCombined achieved a 5× speedup over BDisj. Since BDisj executes each root clause independently of others, additional root clauses meant more tuples materialized multiple times across different root clauses and additional join operators, each with their own overhead. In comparison, TCombined (exec) only experienced modest increases in runtime from executing the additional predicates. The figure also depicts a significant increase in planning time for TCombined for more root clauses. This increase was due to the unoptimized nature of TPullup; recall that TPullup attempts to pull up every filter node one node at a time and see if that results in a plan. As the number of root clauses increased, so did the number of filter nodes, and the number of plan comparisons increased exponentially. A more optimized version of the planner which pulls filter nodes up to the next join juncture could substantially decrease planning time. The plot for the CNF version of the query also showed an increasing difference in the runtime between the BPushConj and TCombined (exec) (TCombined was faster), though not as severe as Figure 4c. Since BPushConj performs all joins before performing any filter operations, even with the base 2 root clauses, there was a large difference in runtime between BPushConj and TCombined, and increasing the number of root clauses only had a secondary effect.

Outer Conjunctive Factor. Throughout our experiments, we observed that if there were selective predicate subexpressions which filtered out most of the tuples early in the pipeline, tagged execution would not have a chance to exhibit its benefits. In an attempt to isolate and measure this factor, we conducted an experiment in which we varied the selectivity of an additional conjunctive predicate expression, henceforth termed the *outer conjunctive factor*. For CNF queries, the predicate

expression with an outer conjunctive factor of 0.1 had the form: `T0.A1 < 0.1 AND (T1.A1 < 0.2 OR T2.A1 < 0.2) AND (T1.A2 < 0.2 OR T2.A2 < 0.2)`. For DNF queries, the same `T0.A1 < 0.1` was included in each root clause. Figure 4d shows the results for CNF queries. As we can see, the runtimes for BPushConj and TCombined remain mostly similar until a sharp increase when the outer conjunctive factor is 0.6. At this point, the tagged execution model finally had a chance to display its benefits, and the difference in runtime peaked when the outer conjunctive factor was 1.0, with TCombined achieving a 10× speedup over BPushConj. The reason for the sharp increase at 0.6 was because the first record in `T0` (i.e., `T0.id = 1`) had a `A1` value between 0.5 and 0.6. As mentioned, we used a Zipf distribution to generate foreign keys randomly, and the most common value that appears in a Zipf distribution is 1. Thus, the inclusion of the first record when the outer conjunctive factor rose to 0.6 meant a substantial increase in the resulting output and a substantial increase in the runtime for BPushConj. Although the same was true for TCombined, this inclusion had a much smaller impact on tagged execution due to its efficient join operators with selective tag maps. The plot for DNF queries had the same shape, for the same reasons.

6 RELATED WORK

The set of related work can largely be divided into two parts: (1) works related to tagging (2) works related to disjunctions.

Tagging. Though not explicit, the idea of tagging is present in past works. Most works in shared work optimization [6] [9] [10] [3] [25] [16] use some form of tags to keep track of which tuples belong to which queries, and works built on the eddy processor [2] [28] [13] typically use a tag to keep track of which operators have been evaluated for each tuple. However, the context in which tags are used for these works is clearly very different compared to our work. In addition, another critical difference between these works and tagged execution is the *semantic* information present in the tags for tagged execution. The tags used in these works all represent some sort of simple membership into a query/operator set, and it is difficult to see that as semantic information compared to the predicate results present in the tags for tagged execution.

Disjunctions (Bypass). The most relevant related work is the line of work regarding the bypass technique [22] [34] [7]. As originally introduced by Kemper et al. [22], the bypass technique augments filter operators in traditional execution with an optional "false" output stream. Input tuples which evaluate to true are output to the regular "true" output stream, and those that evaluate to false are sent to the "false" output stream. The desire is that the tuples in the "true" output stream bypass the more expensive filter operators that appear later in the pipeline. Steinbrunn et al. [34] extend this idea to join conditions, and Claussen et al. [7] expand the technique to include NULL values. While similar, tagged execution different from bypass in two main ways: (1) First is the reuse of query operators and shared work. By using tags to encode predicate expression results, tagged execution can reuse the same query operators for different relational slices with different predicate expression results. However, bypass depends on traditional relational query operators, and this results in using different query operators for relations with different predicate expression results. Thus, even for a query as simple as Query 1, bypass requires multiple scans of the input tables and multiple join operators to execute. (2) Second is the separation of tag space and query plan space. Tagged execution encapsulates the evaluation of the predicate predicate in tags and separates this tag space from the query plan. As a result, tagged execution planners can make planning decisions independently of the predicate expression evaluation. On the other hand, bypass embeds the predicate expression evaluation directly into the query plan and only produces

plans in which predicates are all pushed down. However, as discussed in Sections 4 and 5, this is not always desirable.

Disjunctions (CNF/DNF). Aside from the bypass technique, most works involving disjunctive predicate expressions typically focus on converting the predicate expression into either CNF or DNF and optimizing the execution from these forms [33] [36] [18] [31] [4]. BDisj and BPushConj in our evaluation serve as representatives for these works. However, in addition to the inefficiencies of CNF/DNF-based methods highlighted in the introduction, it is well-known that conversion into CNF/DNF can result in an exponential number of terms [32], so just transforming the input into the correct form can be quite expensive. In comparison, tagged execution has no such requirements on the form of the input predicate expression and optimizes all predicate expression forms equally well.

Disjunctions (Factorization). Chaudhuri et al. [5] introduce techniques to factorize a disjunctive predicate expression to maximize usage of existing indexes. In a sense, factorization works in the "opposite" direction of tag generalization, and we could have used factorization-like techniques to reduce the tag space in our work. However, the techniques presented by Chaudhuri et al. involve searching over an exponential space, as opposed to GeneralizeTag's linear runtime, and tag generalization has the highly convenient benefit of exposing which tags can be discarded early.

Disjunctions (Ordering). The final group of works on disjunctions all have to do with ordering predicates [12] [20] [21] [23]. Given a query with a disjunctive predicate expression, these works attempt to find the best order to evaluate the predicates of that predicate expression. However, these works all assume a single-table query with no joins, so their relevance to our work is limited.

7 CONCLUSION

In this paper, we presented the tagged execution model, a powerful new way to execute queries, and demonstrated how it can be used to optimize disjunctive queries. We showed how we could reduce the tag space using tag generalization and introduced several new planners which can take advantage of the benefits offered by tagged execution. Finally, in our evaluation, we showed that our tagged execution planners outperform traditional execution planners with an average speedup of 2.7× and a maximum speedup of 19× in certain situations, while only incurring an average overhead of 10%, highlighting the prowess of the tagged execution model.

REFERENCES

[1] Laurenz Albe. 2018. Avoid OR for Better PostgreSQL Query Performance. https://www.cybertec-postgresql.com/en/avoid-or-for-better-performance/

[2] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. ACM, Dallas Texas USA, 261–272. https://doi.org/10.1145/342009.335420

[3] George Candea, Neoklis Polyzotis, and Radek Vingralek. 2009. A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB)*. https://infoscience.epfl.ch/record/139392

[4] Jae-young Chang and Sang-goo Lee. 1997. An Optimization of Disjunctive Queries: Union-Pushdown. In *COMPSAC*. IEEE Computer Society, 356–361.

[5] Surajit Chaudhuri, Prasanna Ganesan, and Sunita Sarawagi. 2003. Factorizing Complex Predicates in Queries to Exploit Indexes. In *SIGMOD 2003*.

[6] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. 2000. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. ACM, Dallas Texas USA, 379–390. https://doi.org/10.1145/342009.335432

[7] Jens Claussen, Alfons Kemper, Guido Moerkotte, Klaus Peithner, and Michael Steinbrunn. 2000. Optimization and Evaluation of Disjunctive Queries. *IEEE Transactions on Knowledge and Data Engineering* 12, 2 (2000), 238–260. https://ieeexplore.ieee.org/abstract/document/842265/

[8] Marcus Fontoura, Suhas Sadanandan, Jayavel Shanmugasundaram, Sergei Vassilvitski, Erik Vee, Srihari Venkatesan, and Jason Zien. 2010. Efficiently Evaluating Complex Boolean Expressions. In *Proceedings of the 2010 International Conference on Management of Data - SIGMOD '10*. ACM Press, Indianapolis, Indiana, USA, 3. https://doi.org/10.1145/1807167.1807171

[9] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2012. SharedDB: Killing One Thousand Queries With One Stone. *Proceedings of the VLDB Endowment* 5, 6 (2012). http://www.vldb.org/pvldb/vol5/p526_georgiosgiannikis_vldb2012.pdf

[10] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. 2014. Shared Workload Optimization. *Proceedings of the VLDB Endowment* 7, 6 (Feb. 2014), 429–440. https://doi.org/10.14778/2732279.2732280

[11] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. 2010. HYRISE: A Main Memory Hybrid Storage Engine. *Proceedings of the VLDB Endowment* 4, 2 (2010), 105–116.

[12] Michael Z. Hanani. 1977. An Optimal Evaluation of Boolean Expressions in an Online Query System. *Commun. ACM* 20, 5 (May 1977), 344–347. https://doi.org/10.1145/359581.359600

[13] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Samuel Madden, Vijayshankar Raman, and Mehul A. Shah. 2000. Adaptive Query Processing: Technology in Evolution. *IEEE Data Eng. Bull.* 23, 2 (2000), 7–18. http://www.cs.cmu.edu/~natassa/courses/15-721/papers/hellerstein.pdf

[14] Joseph M. Hellerstein and Michael Stonebraker. 1993. Predicate Migration: Optimizing Queries with Expensive Predicates. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. 267–276.

[15] Justus Henneberg, Felix Schuhknecht, Philipp Reutter, Nils Brast, and Peter Spichtinger. 2022. Northlight: Declarative and Optimized Analysis of Atmospheric Datasets in SparkSQL. In *Proceedings of the 34th International Conference on Scientific and Statistical Database Management*. 1–12.

[16] Mingsheng Hong, Mirek Riedewald, Christoph Koch, Johannes Gehrke, and Alan Demers. 2009. Rule-Based Multi-Query Optimization. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. 120–131.

[17] S. Tonia Hsieh and George V. Lauder. 2004. Running on Water: Three-dimensional Force Generation by Basilisk Lizards. *Proceedings of the National Academy of Sciences* 101, 48 (2004), 16784–16788.

[18] JarkeMatthias and KochJurgen. 1984. Query Optimization in Database Systems. *ACM Computing Surveys (CSUR)* (June 1984). https://dl.acm.org/doi/abs/10.1145/356924.356928

[19] Shuping Ji and Hans-Arno Jacobsen. 2021. A-Tree: A Dynamic Data Structure for Efficiently Indexing Arbitrary Boolean Expressions. In *Proceedings of the 2021 International Conference on Management of Data*. 817–829.

[20] Fisnik Kastrati and Guido Moerkotte. 2017. Optimization of Disjunctive Predicates for Main Memory Column Stores. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 731–744. https://doi.org/10.1145/3035918.3064022

[21] Fisnik Kastrati and Guido Moerkotte. 2018. Generating Optimal Plans for Boolean Expressions. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 1013–1024. https://doi.org/10.1109/ICDE.2018.00095

[22] Alfons Kemper, Guido Moerkotte, Klaus Peithner, and Michael Steinbrunn. 1994. Optimizing Disjunctive Queries with Expensive Predicates. In *ACM SIGMOD Record*, Vol. 23. ACM, 336–347.

[23] Albert Kim, Atalay Mert Ileri, and Sam Madden. 2022. Optimizing Query Predicates with Disjunctions for Column Stores. arXiv:2002.00540 [cs] http://arxiv.org/abs/2002.00540

[24] Albert Kim and Samuel Madden. 2024. Optimizing Disjunctive Queries with Tagged Execution. arXiv:2404.09109 [cs] http://arxiv.org/abs/2404.09109

[25] Sailesh Krishnamurthy, Michael J. Franklin, Joseph M. Hellerstein, and Garrett Jacobson. 2004. The Case for Precision Sharing. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases-Volume 30*. 972–984.

[26] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-store 7 Years Later. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1790–1801.

[27] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[28] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. 2002. Continuously Adaptive Continuous Queries over Streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. 49–60.

[29] Jim Melton and Alan R. Simon. 1993. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann. https://books.google.com/books?hl=en&lr=&id=ZOOMSTZ4T_QC&oi=fnd&pg=PA1&dq=Understanding+the+New+SQL:+A+Complete+Guide&ots=e3HaeNQ5hu&sig=7hniGDnnji36apNp8_6hvtt5VtY

[30] Murali Muralikrishna. 1988. *Optimization of Multiple-Disjunct Queries in a Relational Database System*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[31] M. Muralikrishna and David J. DeWitt. 1988. Optimization of Multiple-Relation Multiple-Disjunct Queries. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems - PODS '88*. ACM Press, Austin, Texas, United States, 263–275. https://doi.org/10.1145/308386.308452

[32] Stuart J. Russell and Peter Norvig. 2016. *Artificial Intelligence: A Modern Approach*. Malaysia; Pearson Education Limited,.

[33] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data - SIGMOD '79*. ACM Press, Boston, Massachusetts, 23. https://doi.org/10.1145/582095.582099

[34] Michael Steinbrunn, Klaus Peithner, Guido Moerkotte, and Alfons Kemper. 1995. Bypassing Joins in Disjunctive Queries. In *VLDB*, Vol. 95. 11–15.

[35] Michael Stonebraker and Lawrence A. Rowe. 1986. *The Design of Postgres*. Vol. 15. ACM.

[36] David D. Straube and M. Tamer Özsu. 1990. Queries and Query Processing in Object-Oriented Database Systems. *ACM Transactions on Information Systems (TOIS)* 8, 4 (1990), 387–430.

Received October 2023; revised January 2024; accepted February 2024