Department of Informatics, University of Zürich

BSc Thesis

A General-purpose Range Join Algorithm for PostgreSQL

Thomas Rolf Mannhart

Matrikelnummer: 17-917-907

Email: thomasrolf.mannhart@uzh.ch

March 16, 2020

supervised by Prof. Dr. M. H. Böhlen and Prof. Dr. A. Dignös

University of Zurich[UZH]

Department of Informatics

Acknowledgements

I would like to offer my special thanks to my supervisor, Prof. Dr. Anton Dignös, with whom it was and will be a pleasure to work.

I would like to thank, Prof. Dr. Michael H. Böhlen and the Database Technology Group of the University of Zurich for making this thesis possible.

I would also like to thank the Database Systems Group of the Free University of Bozen-Bolzano, where I felt very welcome.

Abstract

In this thesis we provide a range join algorithm based on the sort-merge paradigm and its implementation into the open-source RDBS PostgreSQL. The traditional sort-merge join is an efficient join algorithm for equality constraints, while a range join additionally considers a predicate describing that a value from one relation is in the range between two values of the other relation. PostgreSQL implements the sort-merge join or Merge Join (MJ) as a state machine adhering to the demand-pull pipeline paradigm. Our range join or Range Merge Join (RMJ) builds on the existing implementation and expands it with additional conditions to efficiently handle range joins. We describe in detail, how we modified the PostgreSQL optimizer and executor to achieve this goal. We provide the implementation of the RMJ algorithm as well as the identification of possible range join predicates and the correct sorting of the input relations. We show the benefits of our implementation in several experiments using real-world and synthetic workloads and datasets. The experiments show a major reduction in execution time in most real-world and all of our synthetic workloads, while only incurring a minor overhead in planning time in a few cases.

Zusammenfassung

In dieser Thesis zeigen wir einen Range-Join-Algorithmus, der auf dem Sort-Merge-Paradigma basiert und dessen Implementierung im Open-Source-RDBS PostgreSQL. Der Sort-Merge-Join ist ein effizienter Join-Algorithmus für Gleichheitsbedingungen. Ein Range-Join berücksichtigt ein zusätzliches Prädikat, das beschreibt, dass ein Wert aus einer Relation im Bereich zwischen zwei Werten der anderen Relation liegt. PostgreSQL implementiert den Sort-Merge-Join oder Merge-Join (MJ) als eine "state machine", welche nach dem Demand-Pull-Pipelining-Prinzip funktioniert. Unser Range-Join oder Range-Merge-Join (RMJ) baut auf der bestehenden Implementierung auf und erweitert sie um zusätzliche Bedingungen, um Range-Joins effizient zu handhaben. Wir beschreiben im Detail, wie wir den Optimizer und Executor von PostgreSQL modifiziert haben. Wir zeigen nicht nur die Implementierung des RMJ-Algorithmus, sondern auch die Identifizierung möglicher Range-Join-Prädikate und die korrekte Sortierung der Input-Relationen. Wir zeigen die Vorteile unserer Implementierung in mehreren Experimenten unter Verwendung von realen und synthetischen Arbeitslasten und Datensätzen. Die Experimente zeigen eine erhebliche Reduzierung der Ausführungszeit in einigen realen und all unseren synthetischen Arbeitslasten mit nur einem geringfügig höheren Planungsaufwand in bestimmten Fällen.

Contents

1. Introduction **1**
1.1. Motivation . . . 1
1.2. Contributions . . . 3
1.3. Organization of the Thesis . . . 3

2. Background **4**
2.1. Sort-Merge Join . . . 4
2.2. Range Join . . . 5

3. A Range Merge Join in PostgreSQL **6**
3.1. The Sort-Merge Join Implementation in PostgreSQL . . . 6
3.2. A Range Merge Join Implementation for PostgreSQL . . . 8
3.2.1. Algorithm . . . 8
3.2.2. Implementation as a State Machine . . . 9

4. Integration into the PostgreSQL Kernel **12**
4.1. Overview . . . 12
4.2. Path Nodes . . . 12
4.3. Planner/Optimizer . . . 13
4.3.1. range clauses . . . 13
4.3.2. Sorting . . . 15
4.4. Executor . . . 17
4.4.1. Initialization . . . 17
4.4.2. Execution . . . 17
4.5. Query Plans . . . 18

5. Experiments **20**
5.1. Setup . . . 20
5.1.1. Synthetic Workloads . . . 20
5.1.2. Real-World Workloads . . . 21
5.2. Overview . . . 21
5.3. Overhead of the RMJPATCH . . . 22
5.3.1. Workload and Methodology . . . 22
5.3.2. Planning Time . . . 22
5.3.3. Execution Time . . . 24

5.4. Range Join Execution Time . . . 25
5.4.1. RMJ Vs. MJ . . . 25
5.4.2. Merge Condition Vs. Range Condition . . . 26
5.4.3. RMJPATCH Vs. HEAD . . . 27
5.4.4. RMJ Vs. Index Joins . . . 28
5.4.5. Real-World Workloads . . . 30
5.5. Summary . . . 32

6. Experiences from the PostgreSQL Implementation 33
6.1. Procedure . . . 33
6.2. Takeaways . . . 34
6.3. Open Issues . . . 35

7. Conclusion and Future Work 37

A. Appendix 40
A.1. TPC-H Query Description . . . 40
A.2. TPC-H Planning Time Without Filtering Constants . . . 41
A.3. Synthetic Experiments With Parallelization Enabled . . . 42

List of Figures

1.1. Example input relations **marks** and **grades** 1
1.2. Result of range join between **marks** and **grades** 2
1.3. Example input relations **emps** and **events**. 2
1.4. Result of range join between **emps** and **events** with an equality condition on attribute dept. 3

3.1. State diagram of PostgreSQL's sort-merge join implementation. 8
3.2. State diagram for the range merge join implementation as an extension of the sort-merge join. 10

4.1. Query plans for the introductory examples in Chapter 1.1. 19

5.1. Planning time for varying number of joins and conditions. 24
5.2. Runtime results for our synthetic workloads with only MJ and RMJ enabled. . 26
5.3. Runtime results for varying selectivity κ or δ. 27
5.4. Runtime results for our synthetic workloads with all join algorithms enabled. 28
5.5. Runtime results for our synthetic workload with range conditions only (smaller default values for input relations, $n = m =$ 10k). 29
5.6. Runtime results for our synthetic workload with range conditions only and a B+-Tree index on s for HEAD (smaller default values for input relations, $n = m =$ 1M). 30
5.7. Runtime results for our synthetic workload with range conditions only and a GiST or SP-GiST index on **r** for HEAD (smaller default values for input relations, $n = m =$ 1M). 31
5.8. RMJPATCH vs. HEAD on real-world workload. 32

6.1. Simplified state diagrams of the PostgreSQL Merge Join state machine. . . . 33

A.1. Runtime results for our synthetic workloads with only MJ and RMJ enabled. . 42
A.2. Runtime results for our synthetic workloads with all join algorithms enabled. 43
A.3. Runtime results for our synthetic workload with range conditions only (smaller default values for input relations, $n = m =$ 10k). 44
A.4. Runtime results for our synthetic workload with range conditions only and a B+-Tree index on s for HEAD (smaller default values for input relations, $n = m =$ 1M). 45

A.5. Runtime results for our synthetic workload with range conditions only and a GiST or SP-GiST index on $\mathbf{r}$ for HEAD (smaller default values for input relations, $n = m = 1$M). 46

List of Tables

3.1. States of the PostgreSQL Merge Join 6

5.1. Parameters of synthetic data used in the experiments. 20
5.2. Notation for statistical significance. 22
5.3. Planning time in milliseconds for the TPCH queries 23
5.4. Execution time in milliseconds for the TPCH queries 25

A.1. Description of TPC-H queries. 40
A.2. Planning time in milliseconds for the TPCH queries (without filtering constants). 41

1. Introduction

1.1. Motivation

Joins are arguably one of the most important, frequent, and expensive operations in relational database systems (RDBS). Traditionally, joins are mostly based on equality constraints (i.e., equi joins), such as key-foreign-key joins. This type of joins with an equality join condition are well supported in contemporary RDBSs and efficient evaluation techniques exist, such as hash join [17], sort-merge join [19], or index join [18].

In some application scenarios joins are based on range conditions, where a value of one relation has to be joined into a range defined by the other relation. This type of join is called *Range Join*. For example, to associate data of a click stream with its origin country [12], it is necessary to join the IP addresses of the click stream with the (several) IP address ranges of the different countries. Other examples can be found in taxation, insurance, and shipping applications [8, 9], where prices are often related to ranges. In a shipping scenario it may be the case, that the weight of a package falling within a specific range, results in a certain price. Range joins are based on inequalities for which most RDBSs fail to provide an efficient evaluation. In most cases the only available join algorithm for range conditions is either a nested loop or an index join, of which former is a brute force approach and latter is only efficient for very selective joins.

Example 1 (Range Join) *Consider the example relations in Figure 1.1. Relation* marks *records the marks of students achieved in an exam. The first tuple records that the student Anton with student number 1232 achieved a mark of 23.5 out of 100 in the exam. Relation* grades *records the grading scheme for the exam. Here, the first tuple records that a student who achieved a mark between 0 and 18 receives a grade of 1. A student that achieved a mark between 18.5 and 36 receives a grade of 2.*

marks

name	snumber	mark
Anton	1232	23.5
Thomas	4356	95
Michael	1125	72
Hans	3425	90

grades

mmin	mmax	grade
0.0	18	1
18.5	36	2
36.5	54	3
54.5	72	4
72.5	90	5
90.5	100	6

Figure 1.1.: Example input relations marks and grades

To determine the grade of each student the two tables need to be joined on the mark that falls between mmin and mmax. This can be done using the following query.

```
SELECT name, snumber, grade
FROM marks JOIN grades ON mark BETWEEN mmin AND mmax;
```

The result of this query is shown in Figure 1.2. For instance, the student with name Anton receives a grade of 2, because his mark of 23.5 falls into the range $[18.5, 36]$ *which corresponds to the grade 2 (cf. relation* grades *in Figure 1.1).*

name	snumber	grade
Anton	1232	2
Thomas	4356	6
Michael	1125	4
Hans	3425	5

Figure 1.2.: Result of range join between marks and grades

In some scenarios range conditions in joins, such as in Example 1, do not occur in isolation, but may also be accompanied with one or more additional equality conditions.

Example 2 (Range Join with an equality condition) *Consider the example relations in Figure 1.3. Relation* emps *records the contracts of employees in departments of a company. The first tuple in* emps*, for instance, records that the employee Anton works in the Sales department during the time period January 2020 to March 2020. Relation* events *records events of the company for which employees of a specific department are required. In* events*, the first tuple records a fair in Switzerland for which an employee of the Marketing department is required on March 5, 2020.*

emps

name	dept	ts	te
Anton	Sales	2020-01-01	2020-03-31
Thomas	Marketing	2020-01-01	2020-06-30
Michael	Marketing	2020-03-01	2020-12-31
Hans	Sales	2020-01-01	2020-12-31
Thomas	Accounting	2020-07-01	2020-12-31

events

event	dept	t
Fair CH	Marketing	2020-03-05
Presentation	Sales	2020-06-15
Fair IT	Marketing	2020-08-03
Balance Report	Accounting	2020-08-03
Product launch	Marketing	2020-10-15

Figure 1.3.: Example input relations emps and events.

To find the employees that are available for an event the two tables need to be joined on the department attribute, as well as on the event date that falls into the employee's contract period. This can be done using the following query.

```
SELECT name, em.dept, event, t
FROM emps em JOIN events ev ON em.dept = ev.dept AND t BETWEEN ts AND te;
```

name	dept	event	t
Thomas	Marketing	Fair CH	2020-03-05
Michael	Marketing	Fair CH	2020-03-05
Michael	Marketing	Fair IT	2020-08-03
Michael	Marketing	Product launch	2020-10-15
Hans	Sales	Presentation	2020-06-15
Thomas	Accounting	Balance Report	2020-08-03

Figure 1.4.: Result of range join between emps and events with an equality condition on attribute dept.

The result of this query is shown in Figure 1.4. For instance, the first tuple records that Thomas from the Marketing department is available for the fair in Switzerland on March 5, 2020. Thomas is not available for other events for the Marketing department as he moved to the Accounting department later on.

The goal of this thesis is to provide an efficient evaluation algorithm for range joins and to implement it into the open-source RDBS PostgreSQL.

1.2. Contributions

The technical contributions of this thesis are as follows:

- We provide an algorithm for range joins based on the well known sort-merge paradigm and transform it to fit the demand-pull pipelining mechanism adopted by most RDBS.
- We show an implementation of our algorithm as an extension of the sort-merge join execution algorithm, termed range merge join, and its integration into the analyzer, optimizer, and executor of the open-source RDBS PostgreSQL.
- We conduct extensive experiments on synthetic and real-world workloads to show the efficiency of our implementation as well as the tight integration into the kernel of PostgreSQL.

1.3. Organization of the Thesis

The remainder of this thesis is organized as follows: Chapter 2 introduces the necessary background on sort-merge join algorithm and range joins; Chapter 3 provides the implementation of the range merge join execution algorithm and its variant using a demand-pull pipelining mechanism; Chapter 4 shows the integration of our range merge join into the optimizer, and executor of the open-source RDBS PostgreSQL; Chapter 5 reports the results of our experimental evaluation on synthetic and real-world workloads; Chapter 6 describes our experience working on the PostgreSQL kernel; Chapter 7 concludes the thesis and provides directions for future work.

2. Background

In this section we first give the necessary background on the sort-merge join [19] on which the implementation of our range merge join is based on, and then formally define the range join.

2.1. Sort-Merge Join

The sort-merge join [19] (MJ) is a join algorithm for equality conditions that requires both input relations to be sorted by their join attribute. If the input relations are not sorted, an explicit sort step is performed before the join. The algorithm for the sort-merge join is shown in Algorithm 1. The main idea of the MJ is as follows. The input relations of the join are

Algorithm 1: MJ$(\mathbf{r}, \mathbf{s}, E)$

Input: Relations $\mathbf{r}$ and $\mathbf{s}$ and equality attributes E.
Output: Result of $\mathbf{r} \bowtie_{\mathbf{r}.E=\mathbf{s}.E} \mathbf{s}$.

```
1  r ← first(r);
2  s ← first(s);
3  while r ≠ ω ∧ s ≠ ω do
4      if r.E < s.E then
5          r ← next(r);                                  // skip outer
6      else if r.E = s.E then
7          marked ← s;                                   // mark
8          while s ≠ ω ∧ r.E = s.E do
9              output r and s;
10             s ← next(s);
11         r ← next(r);                    // end of matches for outer
12         if r ≠ ω ∧ r.E = marked.E then
13             s ← marked;                               // backtrack inner
14     else
15         s ← next(s);                                  // skip inner
```

scanned from the beginning until at least one relation finishes during the join process. For each tuple of the outer relation, one of three cases may apply: (i — lines 4–5) the current outer tuple is smaller in sort order than the inner tuple, in this case the current outer tuple is skipped; (ii — lines 6–13) the current outer and inner tuples have the same join attributes, in this case the inner tuple is marked for possible other outer tuples, a join match is produced, and all subsequent tuples in the inner relation are checked and matched. The outer relation is

advanced and if the new tuple has the same join attributes as the marked inner tuple, the inner relation is backtracked; (iii — lines 14–15) the current outer tuple is larger in sort order than the inner tuple, in this case the inner tuple is skipped.

The sort-merge join is an effective algorithm for joining on equality conditions with complexity $\mathcal{O}(n+m+z)$ for sorted relations, where n and m are the size of the two input relations respectively, and z is the size of the result.

2.2. Range Join

As the goal of this thesis is to implement a range join in PostgreSQL, we here report its formal definition. A *range join* is a join in which the join predicate specifies that a value from one relation is in the range between two values defined by the other relation.

Definition 1 (Range Join) *Let* $\mathbf{r}$ *and* $\mathbf{s}$ *be relations with schema* R *and* S*, respectively;* $E \in R \cap S$ *be a set of joint attributes; attributes* $ts \in R$ *and* $te \in R$ *represent an interval in* $\mathbf{r}$*; and attribute* $t \in S$ *be an attribute with the same domain as* ts *and* te*. Let further be* $\prec^s \in \{<, \leq\}$ *and* $\prec^e \in \{<, \leq\}$*. A* range join *between* $\mathbf{r}$ *and* $\mathbf{s}$ *is expressed as follows:*

$$\mathbf{r} \bowtie_{\mathbf{r}.E=\mathbf{s}.E \wedge \mathbf{r}.ts \prec^s \mathbf{s}.t \prec^e \mathbf{r}.te} \mathbf{s}$$

The two comparison operators $\prec^s \in \{<, \leq\}$ and $\prec^e \in \{<, \leq\}$ define whether t can be equal to ts and/or te respectively.

3. A Range Merge Join in PostgreSQL

3.1. The Sort-Merge Join Implementation in PostgreSQL

PostgreSQL adopts a demand-driven or demand-pull pipelining mechanism [14]. Each execution algorithm for an operator in a query returns the next tuple to the caller (execution algorithm preceding it) until there are no more tuples to return in which case it returns NULL [15].

To adhere to this mechanism, the sort-merge join is implemented as a state machine shown in Figure 3.1. This algorithm implements the MJ of Algorithm 1, but note that while Algorithm 1 implements an inner join, this state machine implementation also includes an early stop mechanism for anti joins, and a mechanism for outer joins.

Each time the executor function is called, the state machine returns one tuple or NULL to signal the end of the join. The state is preserved as context information in an internal data structure called `MergeJoinState` and passed along to each call of the executor function. Below we describe the transitions in the state machine together with a short description of all the different states in the Table 3.1. For a better visibility we use shorter names for the states, in PostgreSQL the merge join states are prefixed with `EXEC_MJ_`, i.e., `NEXTINNER` corresponds to `EXEC_MJ_NEXTINNER` in PostgreSQL.

Table 3.1.: States of the PostgreSQL Merge Join

State	Description
`INITIALIZE_OUTER`:	Fetch the first outer tuple.
`INITIALIZE_INNER`:	Fetch the first inner tuple.
`JOINTUPLES`:	Join the current outer and inner tuples.
`NEXTINNER`:	Fetch the next inner tuple.
`NEXTOUTER`:	Fetch the next outer tuple.
`TESTOUTER`:	Check if the current outer and the marked inner tuple match. Restore the marked inner if they do. Fetch the next inner otherwise.
`SKIP_TEST`:	Check if the current tuples match and mark the inner tuple if they do.
`SKIPOUTER_ADVANCE`:	Skip the current outer tuple and fetch the next outer tuple.
`SKIPINNER_ADVANCE`:	Skip the current inner tuple and fetch the next inner tuple.
`ENDOUTER`:	In case of a right or full join NULL-fill any unmatched inner tuples.
`ENDINNER`:	In case of a left or full join NULL-fill any unmatched outer tuples.

The initial state of the state machine is `INITIALIZE_OUTER` to fetch the first tuple

from the outer relation. Every time a new tuple is fetched, it is evaluated and flagged as ENDOFJOIN if the fetched tuple is NULL, i.e., we are at the end of this relation; as NONMATCHABLE if the fetched tuple is not able to match anything, for instance if one of the attributes in the equality condition has a NULL value that cannot be equal to any other value; and as MATCHABLE otherwise. If both, the outer and inner tuples are MATCHABLE, their equality (or merge) attributes are compared and the result is stored in $compareResult$. If $compareResult = 0$, the tuples match, i.e. their merge attributes are equal. In this case they can be joined. If $compareResult < 0$, the outer tuple's merge attributes are smaller then the inner tuple's merge attribute and the next outer tuple is fetched. Finally, if $compareResult > 0$, the outer tuple's merge attributes are larger then the inner and we fetch the next inner tuple.

In SKIP_TEST, when two tuples match, the current position of the inner subplan (preceding algorithm in the pipeline) is marked by *ExecMarkPos()* and a copy of the current inner tuple is saved. When a new outer tuple is fetched in NEXTOUTER, it is compared to the marked (copied) inner tuple in TESTOUTER, not the current inner. If they match, the marked inner tuple becomes the new current inner tuple and the inner subplan is restored to the marked position by *ExecRestrPos()*. Otherwise, if the current inner is MATCHABLE, it will be considered in SKIP_TEST.

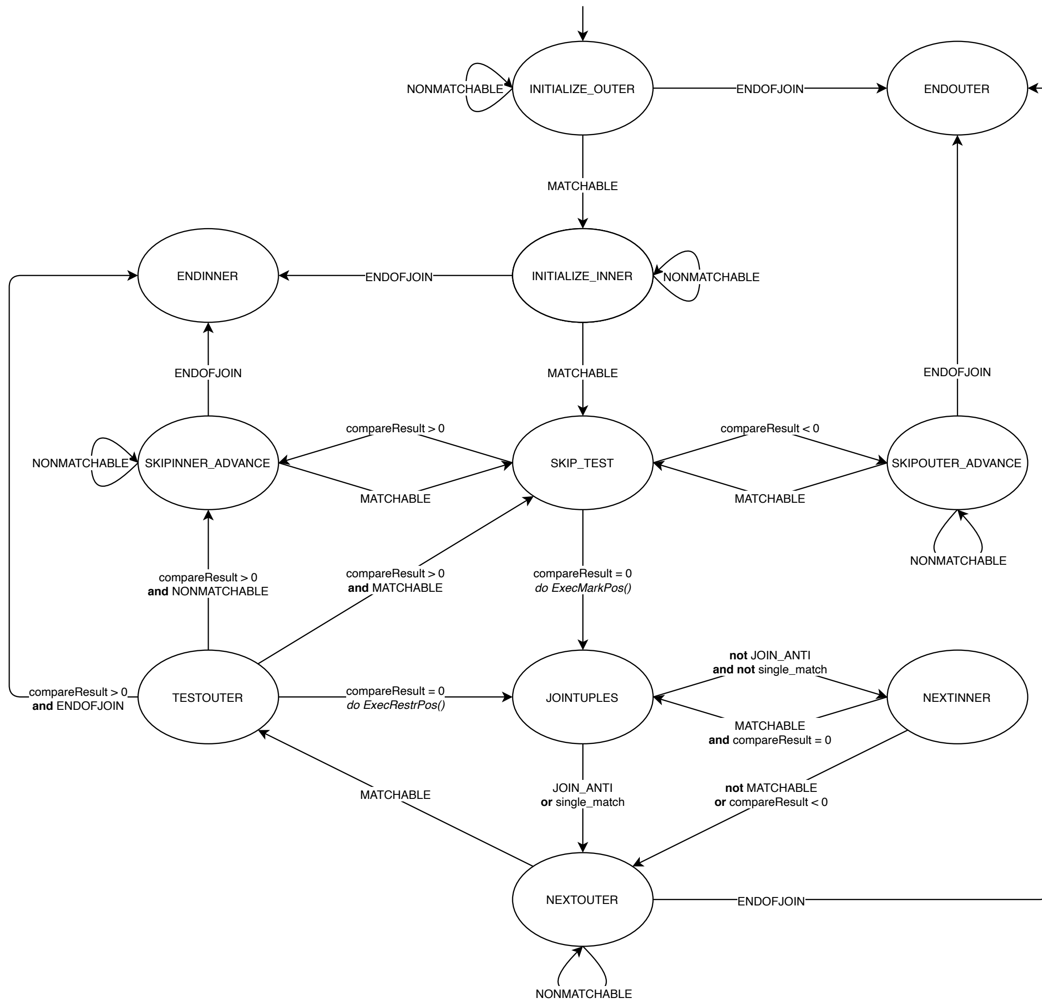


Figure 3.1.: State diagram of PostgreSQL's sort-merge join implementation.

3.2. A Range Merge Join Implementation for PostgreSQL

In this section we report the implementation of our range merge join (RMJ) as an extension of PostgreSQL's sort-merge join (MJ). First, we report its simplified algorithm and later on we show its implementation as an extension to the state machine of the MJ describe above.

3.2.1. Algorithm

The main algorithm for a range inner join is shown in Algorithm 2. It is very similar to the MJ algorithm in Algorithm 1 and requires its input relation $\mathbf{r}$ to be sorted by the attributes of the equality condition E and additionally by the the start of the range condition (ts). The sort order of the other input relation $\mathbf{s}$ is according to the equality condition E and additionally by the value of the range condition (t).

The main differences of the RMJ as compared to the MJ (cf. Algorithm 1) is in the treatment of the range condition. At lines 6 and 8 for a valid join match, the current two input tuples in addition to match on the equality attributes also need to satisfy the range condition $\mathbf{r}.ts \prec^s \mathbf{s}.t \prec^e \mathbf{r}.te$. Similarly, an inner tuple is skipped at lines 14–15 not only when the inner tuples's equality attributes are large in sort order compared to the outer tuple, but also for equal equality attributes and $\neg(r.ts \prec^s s.t)$, i.e., t is before the range defined by the current outer tuple r (and due to the sort order no subsequent outer tuple may have the same equality attributes as r and a range that starts earlier).

Algorithm 2: RMJ($\mathbf{r}, \mathbf{s}, E, \prec^s, \prec^e$)

Input: Relations $\mathbf{r}$ and $\mathbf{s}$, equality attributes E, comparison operator $\prec^s \in \{<, \leq\}$ for start point, and comparison operator $\prec^e \in \{<, \leq\}$ for end point.
Output: Result of $\mathbf{r} \bowtie_{\mathbf{r}.E=\mathbf{s}.E \wedge \mathbf{r}.ts \prec^s \mathbf{s}.t \prec^e \mathbf{r}.te} \mathbf{s}$.

```
1  r ← first(r);
2  s ← first(s);
3  while r ≠ ω ∧ s ≠ ω do
4      if r.E < s.E then
5          r ← next(r);                                        // skip outer
6      else if r.E = s.E ∧ r.ts ≺^s s.t then
7          marked ← s;                                         // mark
8          while s ≠ ω ∧ r.E = s.E ∧ s.t ≺^e r.te do
9              output r and s;
10             s ← next(s);
11         r ← next(r);                      // end of matches for outer
12         if r ≠ ω ∧ r.E = marked.E then
13             s ← marked;                             // backtrack inner
14     else
15         s ← next(s);                                        // skip inner
```

3.2.2. Implementation as a State Machine

Our RMJ is implemented as an extension to state machine of PostgreSQL's existing MJ (cf. Figure 3.1). The extended state machine which enables the execution of range joins is shown in Figure 3.2. The additions to the MJ are highlighted in blue and are preceded by a Boolean flag in the implementation (not shown in the diagram) such that it can be used for both MJ and RMJ by simply setting the flag correspondingly.

If we execute a RMJ, the range condition is only compared, if the merge conditions hold, i.e. $compareResult = 0$. The result of the range comparison is stored in $compareRangeResult$. If $compareRangeResult = 0$, the tuples match, i.e. the inner range attribute is contained in the outer interval. In this case they can be joined. If $compareRangeResult < 0$, the inner range attribute is after the outer interval and the next outer tuple is fetched. Finally, if

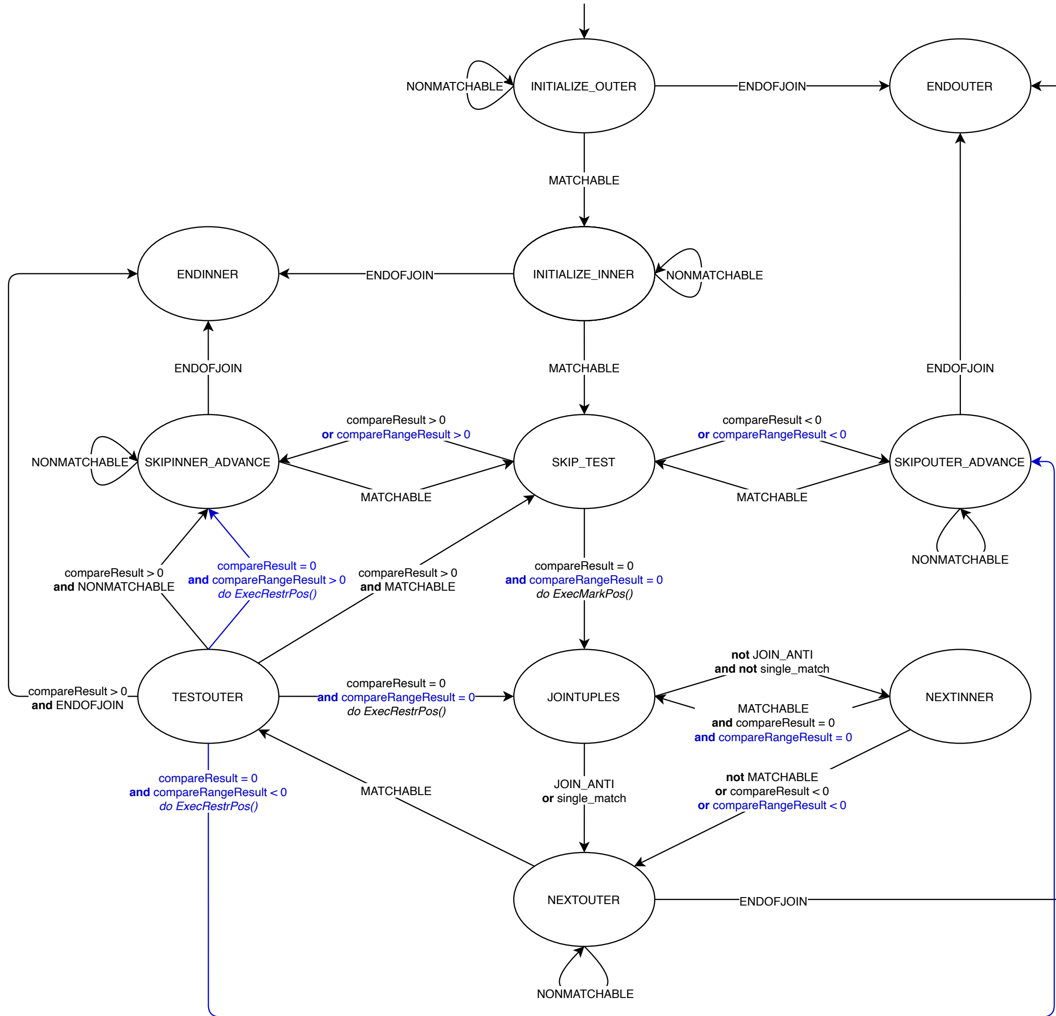


Figure 3.2.: State diagram for the range merge join implementation as an extension of the sort-merge join.

$compareRangeResult > 0$, the inner value is before the outer interval and we fetch the next inner tuple.

Due to the new conditions and transitions some states have a different behaviour (if we execute a RMJ). These altered states are described below.

SKIP_TEST: The current outer and inner tuples equality attributes are compared first. Only if *compareResult* $= 0$, the current tuples range attributes are compared.

If *compareRangeResult* $= 0$, the machine moves to `JOIN_TUPLES`.

If *compareRangeResult* > 0, the machine moves to `SKIPINNER_ADVANCE`.

If *compareRangeResult* < 0, the machine moves to `SKIPOUTER_ADVANCE`.

NEXTINNER: The next inner tuple is fetched and if the inner tuple is `MATCHABLE` the merge attributes are compared. If *compareResult* $= 0$, the range attributes are compared.

If *compareRangeResult* $= 0$, the machine moves to JOIN_TUPLES.

If *compareRangeResult* < 0, the machine moves to NEXTOUTER.

TESTOUTER: The current outer and the marked inner tuples merge attributes are compared. If they match, the marked inner tuple becomes the current inner tuple and the inner subplan is restored to the marked position. Then the range attributes of the current tuples are compared. At this point, the current inner is always the restored marked inner.

If *compareRangeResult* $= 0$, the machine moves to JOIN_TUPLES.

If *compareRangeResult* > 0, the machine moves to SKIPINNER_ADVANCE.

If *compareRangeResult* < 0, the machine moves to SKIPOUTER_ADVANCE.

4. Integration into the PostgreSQL Kernel

4.1. Overview

In PostgreSQL the query processing workflow is composed of several stages: SQL query $\xrightarrow{\text{parser}}$ parse tree $\xrightarrow{\text{analyzer}}$ query tree $\xrightarrow{\text{optimizer}}$ plan tree $\xrightarrow{\text{executor}}$ execution tree.

The integration of the RMJ into PostgreSQL requires modification of two stages and some of their corresponding data structures: the optimizer and the plan tree, and the executor and the execution tree. For each type of tree, additions are introduced that store information required for the processing of the new RMJ algorithm.

In the executor stage of PostgreSQL an execution algorithm consist of three main functions, namely `ExecInit`⟨`Operator`⟩, `Exec`⟨`Operator`⟩, and `ExecEnd`⟨`Operator`⟩ for, respectively, the initialization, the execution, and the finalization of an evaluation algorithm. Here, ⟨`Operator`⟩ is the name of the actual execution algorithm, such as, `MergeJoin`, `Nestloop`, or `Sort`.

At the end of the Sections 4.3.1 – 4.4.2, we describe the methods we introduced to the ProstgreSQL kernel. In most cases, we derived our implementation from existing methods, which is noted at the end of the description.

4.2. Path Nodes

The optimizer creates multiple possible paths to solve a query and uses the best path to build the plan tree. This paths are represented as trees and consist of different types of nodes[1], containing different information.

For the following sections, we need to introduce some of these nodes used by the PostgreSQL's optimizer to build join paths.

`RestrictInfo`: For every logically ANDed restriction condition (in WHERE or JOIN/ON clause), a RestrictInfo is created. They have to be logically ANDed to be able to rule out tuples with only a subset. In our case, these subsets are the merge conditions, the range condition and the join quals. A RestrictInfo node holds important information about the restriction condition, which is used by the optimizer to choose the best query plan.

[1] `https://github.com/postgres/postgres/blob/REL_12_STABLE/src/include/nodes/pathnodes.h`

PathKey: A list of PathKey nodes is used to represent a sort ordering. The first Pathkey represents the primary sort key, the second represents the secondary sort key and so on. An empty PathKey list indicates that there is no particular sort ordering.

EquivalenceClass: In general, EquivalenceClass nodes are created to represent the fact that two or more expressions are equal. In our case, we use EquivalenceClass nodes as the base structure for PathKey nodes. Since every PathKey must reference an EquivalenceClass, we will end up with single-member EquivalenceClass nodes for our range conditions.

4.3. Planner/Optimizer

4.3.1. range clauses

For our RMJ there are two kinds of possible range clauses (range conditions), where we use $\mathbf{r}$ as the outer and s as the inner relation. The first possibility are two inequality conditions of the form $(\mathbf{r}.ts \prec^s \mathbf{s}.t)$ and $(\mathbf{s}.t \prec^e \mathbf{r}.te)$. These two conditions form a range clause and are always in this order, where the start condition is first and the end condition second. The second possibility is a containment condition of the form $(\mathbf{r}.range\ @\!>\mathbf{s}.t)$ or $(\mathbf{s}.t <\!@\ \mathbf{r}.range)$ where $\mathbf{r}$.range is of type rangetype[2] and represents an interval and $\mathbf{s}$.t is an element representing a point. These conditions are read as $(\mathbf{r}.range$ *contains* $\mathbf{s}.t)$ and $(\mathbf{s}.t$ *contained by* $\mathbf{r}.range)$ respectively. These conditions are represented as RestrictInfo nodes, or restrictinfos, as we call them from here on.

The planner creates a list containing all possible range clauses. A range clause itself is a list containing either the two described inequality conditions in the correct order, or one of the above containment conditions. The optimizer will still try the same MJ paths as it would without the RMJ implementation. Additionaly, we try at least one RMJ path for every range clause we constructed in the newly introduced method `select_rangejoin_clauses`.

All restrictinfos which are possible members of a range clause need some preparation. After the initialization of a restrictinfo, the planner recognizes if it is a possible member of a range clause (inequality or containment condition) and we assign the correct operator families[3] (opfamilies) for both arguments to the restrictinfo. To store this information, we expanded the restrictinfo struct with two variables of type `list*` called `rangeleftopfamilies` and `rangerightopfamilies` for the opfamilies of the left and right argument respectively. These opfamilies contain the btree operators of the specific types and are primarily used for sorting (Section 4.3.2), but also to distinguish inequality and containment conditions.

check_rangejoinable: This function checks if a restrictinfo can be part of a range clause and assigns the B+ tree (btree) opfamilies if so.

First we check the `can_join` flag, which indicates that the restrictinfo describes a binary operation expression between two non-overlapping sets of relations, i.e. a possi-

[2] https://www.postgresql.org/docs/12/rangetypes.html
[3] https://www.postgresql.org/docs/12/catalog-pg-opfamily.html

ble candidate for us. Next, we rule out all restrictinfos with clauses containing volatile functions[4].

All restrictinfos that can join and do not contain volatile functions, have to be checked for containment conditions we can use in a range clause. A restrictinfo that represents such a containment condition, has to be handled separately, because `get_rangejoin_opfamilies` will not return any operator families for it.

For such a restrictinfo, we extract the types of both arguments and look them up separately in the type cache[5]. We assign the btree operator families listed in the typecache entries to the restrictinfo.

For any other restrictinfo, we call `get_rangejoin_opfamilies`. If the restrictinfo describes a usable inequality condition, the returned opfamilies are the btree opfamilies for both arguments. Otherwise, NIL is returned and the retrictinfo will not be considered as *rangejoinable*.

This function is derived from `check_mergejoinable` in `initsplan.c`[6], which serves a similar purpose for equality conditions.

`get_rangejoin_opfamilies`: This function checks if an operator is an inequality operator and returns the corresponding btree opfamilies.

We get the operator number (opno) as an input and search the catalog pg_amop[7] to see if the target operator is registered as the "<", "<=", ">" or ">=" operator of any btree opamily. If we find an opfamily, we append it to the result list we return at the end.

This function is the pendant to `get_mergejoin_opfamilies` in `lsyscache.c`[8], which checks for equality operators.

`select_rangejoin_clauses`: This function returns all possible range clauses for a given join.

We get the list of all restrictinfos for the current join. For each restrictinfo, we check first, if the clause is *rangejoinable*, i.e. its `rangeleftopfamilies` are not empty. Next, we check if the restrictinfo is of the correct form. Usable are only restrictinfos of the form "outer op inner" or "inner op outer" where outer and inner are the relations to be joined. If the restrictinfo is usable, we have two possibilities;

The first possibility is that the left and the right operator families of the restrictinfo are equal, which means it describes an inequality condition. In this case we check for all prior inequality conditions in candidates, if there is a possible range clause, combining the two. This is done by calling `range_clause_order`, which also returns the

[4] https://www.postgresql.org/docs/12/xfunc-volatility.html
[5] https://github.com/postgres/postgres/blob/REL_12_STABLE/src/include/utils/typcache.h
[6] https://github.com/postgres/postgres/blob/REL_12_STABLE/src/backend/optimizer/plan/initsplan.c
[7] https://www.postgresql.org/docs/12/catalog-pg-amop.html
[8] https://github.com/postgres/postgres/blob/REL_12_STABLE/src/backend/utils/cache/lsyscache.c

correct order. If it returns an order, we create the range clause and append it to the result list. We append the restrictinfo to the canditadtes list for the following restrictinfos.

The second possibility is a restrictinfo describing a containment condition. It is only considered, if it is of the form "outer @> inner" or "inner <@ outer", because the range has to be an attribute of the outer relation. If this is the case, we create the range clause and append it to the result list.

This function is derived from `select_mergejoin_clauses` in `joinpath.c`[9], which returns all merge clauses for a given join.

`range_clause_order`: This function checks if two restrictinfos can be combined into a range clause and retruns which one is the startclause.

As an input, we get two restrictinfos to compare. We extract all the necessary information from both, i.e. the arguments as nodes, which argument references the outer and which the inner relation and if the strategy is less("<", "<=") or greater (">", ">="). With this information, we decide if these two restrictinfos can be used to create a range clause and which of them describes the start and which the end condition. If the first restrictinfo describes the start condition and consequently has to be listed first, we return 1. Otherwise, if the second has to be listed first, we return 2. If it is not possible to build a range clause, 0 is returned.

4.3.2. Sorting

As mentioned in the previous section, we need the btree opfamilies of the restrictinfos in our range clause to sort the input relations. More specifically, we need the opfamilies of the first restrictinfo in a range clause to build EquivalenceClass nodes, we call eclasses, for both attributes we want to sort. Because we do not know if a restrictinfo will represent the start condition of a range clause or if it will end up as part of a range clause at all, `initialize_range clause_eclasses` creates eclasses for all *rangejoinable* restrictinfos.

These eclasses are the base structure of our PathKey nodes (pathkeys) that define our sort order before merging. After the optimizer defined a sort order for the merge clauses, we step in and add the necessary outer and inner pathkeys for the range clause. If a pathkey already exists, because it belongs to a merge clause, we only use it, if it is at the end of the list. This ensures that the relations are sorted primarily by their equality attributes before they are sorted by their range attributes. This sorting happens in `sort_inner_and_outer` in joinpath.c[10].

For cases where there are no mergclauses, i.e. no merge pathkeys, we try a path for every possible range clause and the corresponding outer and inner pathkeys. This results in n RMJ-paths for n range clauses. Otherwise, the optimzer creates a different sort order for every merge pathkey in the list and creates a normal MJ-path. For m merge clauses, we create $\mathcal{O}(m)$ MJ-paths. For every one of these sort orders generated, we try additional paths for each

[9] `https://github.com/postgres/postgres/blob/REL_12_STABLE/src/backend/optimizer/path/joinpath.c`

[10] `https://github.com/postgres/postgres/blob/REL_12_STABLE/src/backend/optimizer/path/joinpath.c`

range pathkey and the corresponding range clauses, as long as the range pathkey is either at the end of the pathkey list or is not in the list at all. This means, for n range clauses and $m > 0$ merge clauses we create $\mathcal{O}(n * m)$ RMJ-paths, i.e. $\mathcal{O}(m + n * m)$ paths in total.

`initialize_range clause_eclasses`: This function initializes both single-member eclasses of a restrictinfo which has been marked as rangejoinable.

We get a restrictinfo, which already has its range opfamilies assigned. Before we can create the eclasses and assign them to the restrictinfo, we have to extract the types of the attributes.

Usually, these types correspond with the input types of the operator and we retrieve them this way. However, if we deal with a containment condition, we have to extract the type of the element side explicitly, because it is not implied by the operator.

This function is derived from `initialize_merge clause_eclasses` in `pathkeys.c`[11], which initializes the eclasses for equality conditions.

`find_range clauses_for_outer_pathkeys`: This function finds all range clauses that can be used with a set sort order for the outer relation and a specific set of merge clauses.

We take the last merge clause in the list, because it corresponds to the last merge pathkey, i.e. the last pathkey used to sort the outer relation according to the merge clauses. Our range clauses have to correspond to this or the next pathkey, to ensure that the relation is sorted correctly.

For every pathkey, we check if it corresponds to the merge clause. If it does, or the merge clause is NULL, it is a potential range pathkey. Otherwise, we continue the loop with the next pathkey.

We check for every range clause and append the ones who match the pathkey to the result list. If the merge clause is NULL, we either had no merge clauses at all or the previous pathkey corresponded to the merge clause. In this case, we can break the loop. Otherwise we checked the last merge pathkey, we assign NULL to merge clause and loop once more with the next pathkey.

We derived this function from `find_merge clauses_for_outer_pathkeys` in `pathkeys.c`[11].

`select_outer_pathkeys_for_range`: This fuction gets a list of range clauses as input, creates the outer pathkey for every range clause and returns the list of all outer pathkeys.

Similar to `select_outer_pathkeys_for_merge` in `pathkeys.c`[11].

`make_inner_pathkey_for_range`: This function creates and return the inner pathkey for a given range clause.

Equivalent to what `make_inner_pathkey_for_merge` in `pathkeys.c`[11] does for a set of merge clauses.

.

[11] https://github.com/postgres/postgres/blob/REL_12_STABLE/src/backend/optimizer/path/pathkeys.c

4.4. Executor

4.4.1. Initialization

Our RMJ is an extension of the MJ algorithm and data structure and in the initialisation, if there are no range clause, the `mj_RangeJoin` flag is set to false which means that a traditional MJ is performed. If we have a range clause, a new struct called `RangeJoinData` is initialized and filled with the information about the range clause.

`MJCreateRangeData`: This function initializes the `RangeJoinData`, which provides the range clause for the execution.

If the rangclause consists of two inequality conditions, they get initialized as executable expressions and stored in `startClause` and `endClause`. Otherwise, if the range clause consists of a single containment condition, the range argument and the element argument are initialized separately and stored as `rangeExpr` and `elemExpr`. This splitting of the expression is necessary, to be able to use the newly introduced rangetype functions `elem_before_range_internal` and `elem_after_range_internal` during execution.

4.4.2. Execution

During execution, we always check the merge condition first. Only if `MJCompare` returns 0 and we are dealing with a RMJ (indicated by `mj_RangeJoin`), we check the range condition by calling `MJRangeCompare`.

The state machine is implemented as an infinite loop containing a switch statement for the current state, i.e. we have a different case for every state. The first change introduced to the state machine is in the case `EXEC_MJ_NEXTINNER`, where no special cases apply. The second modified part, is in the case `EXEC_MJ_TESTOUTER` where it is to note, that the marked inner relation is always restored if the merge condition holds, regardless of the range condition. This is different in the third and last modified part under `EXEC_MJ_SKIP_TEST`, where the inner relation is only marked, if the merge and range conditions hold.

`MJCompareRange`: This function compares the range attributes of the current tuples and returns 0 if the current tuples match, i.e. the outer intervall contains the inner element, < 0 if the element is after the intervall and > 0 if it is before.

In case of having a `startClause` and an `endClause`, we can just execute them to check if they hold. We check the end clause first and set the result to -1 if it does not hold, i.e. the inner element is after the outer interval. If the start clause does not hold (the inner element is before the outer interval), we set the result to 1.

Otherwise, we have a `rangeExpr` for the outer interval and an `elemExpr` for the inner value. To use the internal methods for rangetypes, we have to look up the typecache for the range expression. We check if the element is after the interval by calling

[11] `https://github.com/postgres/postgres/blob/REL_12_STABLE/src/backend/executor/nodeMergejoin.c`

elem_after_range_internal and set the result to -1 if it is. We check if the element is before the interval by calling elem_before_range_internal and if so, set the result to 1.

If the result has not changed, which means the element is contained in the interval and the range condition holds, we return 0.

This function provides the same functionality for the range clause as MJCompare in nodeMergejoin.c[12] does for the merge clauses.

elem_before_range_internal / elem_after_range_internal: This functions check if an element (point, value) is before (or after) a range (interval, period).

First, we deserialize the rangetype to extract the lower and upper bounds. To check if the element is before the range, we have to compare it to the lower bound. A compare result > 0 means, the element is before the range and we return true. We compare the element to the upper bound, to check if it is after the range. Here, the element is after the range if the compare result is < 0. If the result is 0, i.e. the element is equal to the bound (upper or lower), we have to check if the bound is inclusive and only return true, if it is not.

This functions are basically the two parts of range_contains_elem_internal in rangetypes.c[13].

4.5. Query Plans

In this section we show the query plans produced by our implementation. We use the example from the introduction, and we run them on our RMJ implementation. Using the function EXPLAIN(ANALYZE,TIMING FALSE), PostgreSQL gives the query plans shown in Figure 4.1. We used VACUUM and ANALYZE on all four relations. For Example 2, the relations are too small for the RMJ to be beneficial, as you will see in our experiments in Section 5.4.3. For the sake of this example and to get the RMJ query plan, we disable the more efficient hash-join.

In Figure 4.1a, you can see the range condition ((marks.mark >= grades.mmin) AND (marks.mark <= grades.mmax)) and the sort keys grades.mmin and marks.mark. The **grades** relation is only sorted by its range attribute representing the start of the interval.ad Explain a bit the query plan

In Figure 4.1b, you can see the merge condition (em.dept = ev.dept) and the range condition ((ev.t >= em.ts) AND (ev.t <= em.te)). The sort keys are now em.dept, em.ts for **emps** and ev.dept, ev.t for **event**. As intended, the relations are sorted by their equality attributes first and their range attributes second.

[12] https://github.com/postgres/postgres/blob/REL_12_STABLE/src/backend/executor/nodeMergejoin.c

[13] https://github.com/postgres/postgres/blob/REL_12_STABLE/src/backend/utils/adt/rangetypes.c

```
                                         QUERY PLAN
---------------------------------------------------------------------------------------------
 Range Merge Join  (cost=2.22..2.32 rows=3 width=14) (actual rows=4 loops=1)
   Range Cond: ((marks.mark >= grades.mmin) AND (marks.mark <= grades.mmax))
   ->  Sort  (cost=1.14..1.15 rows=6 width=20) (actual rows=6 loops=1)
         Sort Key: grades.mmin
         Sort Method: quicksort  Memory: 25kB
         ->  Seq Scan on grades  (cost=0.00..1.06 rows=6 width=20) (actual rows=6 loops=1)
   ->  Sort  (cost=1.08..1.09 rows=4 width=18) (actual rows=7 loops=1)
         Sort Key: marks.mark
         Sort Method: quicksort  Memory: 25kB
         ->  Seq Scan on marks  (cost=0.00..1.04 rows=4 width=18) (actual rows=4 loops=1)
 Planning Time: 0.116 ms
 Execution Time: 0.056 ms
(12 rows)
```

(a) Joining **grades** and **marks** (Example 1)

```
                                           QUERY PLAN
---------------------------------------------------------------------------------------------
 Range Merge Join  (cost=2.23..2.33 rows=1 width=29) (actual rows=6 loops=1)
   Merge Cond: (em.dept = ev.dept)
   Range Cond: ((ev.t >= em.ts) AND (ev.t <= em.te))
   ->  Sort  (cost=1.11..1.12 rows=5 width=22) (actual rows=5 loops=1)
         Sort Key: em.dept, em.ts
         Sort Method: quicksort  Memory: 25kB
         ->  Seq Scan on emps em  (cost=0.00..1.05 rows=5 width=22) (actual rows=5 loops=1)
   ->  Sort  (cost=1.11..1.12 rows=5 width=24) (actual rows=6 loops=1)
         Sort Key: ev.dept, ev.t
         Sort Method: quicksort  Memory: 25kB
         ->  Seq Scan on events ev  (cost=0.00..1.05 rows=5 width=24) (actual rows=5 loops=1)
 Planning Time: 0.092 ms
 Execution Time: 0.060 ms
(13 rows)
```

(b) Joining **emps** and **events** (Example 2)

Figure 4.1.: Query plans for the introductory examples in Chapter 1.1.

5. Experiments

5.1. Setup

The experiments were run on a machine with an Intel(R) Xeon(R) CPU E5-2667 v3 @ 3.20GHz, 20480KB of cache, 100GB of RAM, and a 64bit Ubuntu SMP GNU/Linux with the kernel version 4.4.0-143-generic. We use PostgreSQL 13devel (`commit 9f87ae38eaffcc7f72c45bfeb79e09dd6e8c2f48` with the default configuration. To measure the execution time of queries, we use `EXPLAIN (ANALYZE, TIMING FALSE)`, which reports the total planning and execution time excluding the time for producing each tuple in each executor node and the overhead of writing and printing the result.

5.1.1. Synthetic Workloads

We use synthetic data to show the effect of varying individual data characteristics on the performance of our Range Merge Join. Unless otherwise specified, Table 5.1 summarizes the parameters for the synthetic data used in the experiments and their default values. n is the size of relation r, i.e., the relation containing the interval, and m is the size of relation s, i.e., the relation containing the point. κ is the reciprocal ($\frac{1}{x}$) of the number of distinct values in the equality predicate. The values for the equality predicate are uniformly distributed, which means that the equality predicate for κ in the join has selectivity κ. The periods and points are uniformly distributed and as a parameter we use the average duration of periods in % of the domain, which means for an average duration of periods δ the range predicate has selectivity δ.

Table 5.1.: Parameters of synthetic data used in the experiments.

Parameter	Description	Default value
n	size of relation r	$10,000,000$
m	size of relation s	$10,000,000$
κ	selectivity of the equality condition	0.001%
δ	selectivity of the range condition	0.001%

The query we evaluate on the synthetic data is a join between two tables with an equality condition on an attribute `g` and a range condition such that `t` in s falls into the range [`ts`, `te`] in r as follows.

```
SELECT *
FROM r JOIN s ON r.g = s.g AND s.t BETWEEN r.ts AND r.te;
```

Since we did not consider a parallelized version of our RMJ yet and because we got results, where the execution time for HEAD was bigger with parallelization then without, we disabled parallelization for the experiments on our synthtic workloads, i.e., we set `max_parallel_workers_per_gather` = 0. The results with parallelization enabled can be seen in the appendix (Section A.3).

5.1.2. Real-World Workloads

As real-world workloads we compute a range (left) join as it is used in the temporal normalization primitive [6, 7, 4, 5] to compute temporal aggregation [10]. We use the real-world datasets and workloads from [6, 7].

The *Incumbents* dataset [11] from the University of Arizona has 83,857 tuples. Each tuple records a job assignment (pcn) for an employee (ssn) over a specific time interval. The data ranges over 16 years and contains 49,195 employees assigned to 38,178 jobs. The interval timestamps are recorded at the granularity of days and have a duration between 1 and 573 days, with an average of approximately 180 days. The *Flight* dataset [3] contains 55,072 tuples. Each tuple records the actual time interval of a flight from a departure airport (fap) to a destination airport (dap). The data ranges over 10 days and contains 559 different departure and 578 different destination airports. The interval timestamps are recorded at the granularity of minutes and have a duration between 25 and 915 minutes, with an average of approximately 128 minutes.

To compute the temporal normalization primitive for temporal aggregation, the database systems needs to perform a range (left) join, and to show the improvements of RMJPATCH for this operation we execute the following query, where g depending on the normalization operation may either be one of the attributes of the datasets or may be omitted.

```
SELECT *
FROM r LEFT OUTER JOIN
    (SELECT g, ts AS t FROM r UNION SELECT g, te AS t FROM r) AS s
       ON r.g = s.g AND r.ts <= s.ts AND s.ts < r.te;
```

5.2. Overview

Every addition to the PostgreSQL optimizer and executor imposes some overhead in planning and execution time, respectively. In the first part of the evaluation we quantify this overhead for the cases when no RMJ is used. In the second part of the evaluation we focus on the RMJ and compare its execution time for various settings with the traditional equality based joins, such as HJ and MJ, and index based joins. We also show how the RMJ is tightly integrated into the PostgreSQL optimizer.

5.3. Overhead of the RMJPATCH

5.3.1. Workload and Methodology

In this section, we quantify the overhead on palling time and execution of our implementation of a new join algorithm on the PostgreSQL system. We use the TPCH benchmark [1] with scale factor 10, i.e., a database of 10GB and compare our version with the HEAD version of PostgreSQL. For queries Q17 and Q20 we use a scale factor of 1, i.e., a database of 1GB, because already in this case they take 2 hours and 1 hour, respectively, to execute. A short descriptions of the queries is provided in Table A.1. We run each query 10 times sequentially alternating between the two versions, and repeat this ten times, resulting in 100 (10×10) executions of each query. For the queries Q17 and Q20 we only run 1×3 executions. Each query is run in an independent session in order to avoid plan caching. For this experiments we disable the hash join by using `set enable_hashjoin = false` in order to see the full overhead of our implementation. In this case all optimization paths of the merge join are explored without pruning in case the hash join seems more promising at an early stage, but also the merge join is used as join algorithm in all queries. To quantify the overhead, we report average planning and execution times for both versions, the absolute and relative differences of the average and the statistical significance based on a two-sided t-test[1] on the 100 observations to test if the two versions have identical average. The notation for the statistical significance is provided in Table 5.2. The lower the p-value the higher the evidence that the averages differ and thus the difference is statistically significant.

Table 5.2.: Notation for statistical significance.

Notation	Explanation
ns	$p > 0.05$
*	$p \leq 0.05$
**	$p \leq 0.01$
***	$p \leq 0.001$
****	$p \leq 0.0001$
-	not enough samples

5.3.2. Planning Time

The results for planning time are shown in Table 5.3. Generally, the time required for planning ranges from below a millisecond to a few milliseconds and the largest absolute overhead of our patch RMJPATCH compared to HEAD is about 0.18 milliseconds. In terms of relative difference, we have the largest overhead for query Q18, which is also statistically significant. Since we encountered many outliers in both systems for theses small numbers we truncated the largest 10% of running and execution time. For the long running queries Q17 and Q20 RMJPATCH has a lower planning time, but since this queries run for several hours, there are

[1] In Python: `scipy.stats.ttest_ind(a=group1, b=group2, equal_var=False)`

not enough samples to provide statistical significance, and in general RMJPATCH does not omit paths in planning so the difference can only be explained by noise and outliers in the running time.

From these experiments, we do not see a clear pattern in overhead neither for many joins nor for inequalities (both parameters, which are inspected for the RMJ during planning). Query Q8 is the query with the most number of joins, i.e., eight joins, and query Q6 the one containing most inequalities, i.e., five inequalities, and for each of this condition our RMJPATCH in contrast to HEAD evaluates whether it can be used as a range condition. In both of these queries the overhead of RMJPATCH is very small.

We repeated the experiments without pruning constants and other unusable expressions in the planning phase, i.e. without checking the `can_join` flag (cf. Section 4.3.1). The results can be seen in Table A.2 in the Appendix. In this case the overhead for the queries with inequalities would be high (see Q6, Q8, and Q14 in Table A.2). Once the filtering of constants in the inequality conditions was implemented, inequalities seem not to cause high overhead.

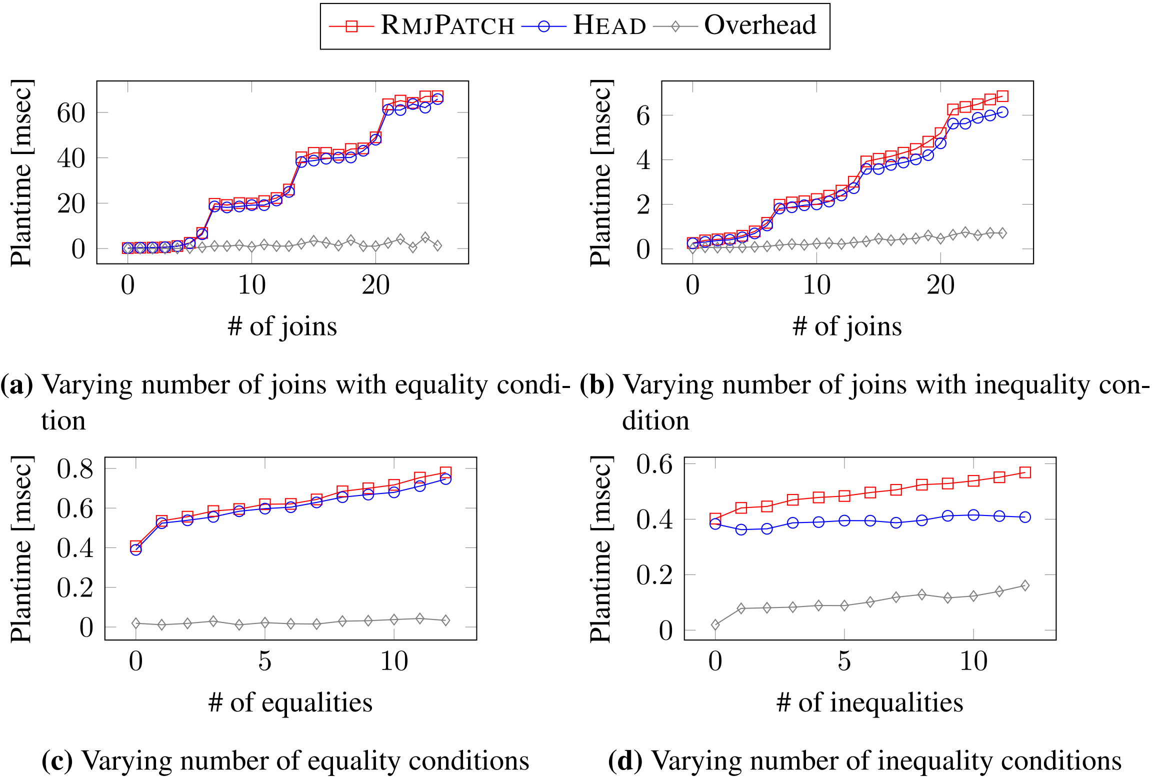
Table 5.3.: Planning time in milliseconds for the TPCH queries

Query	RMJPATCH	HEAD	Difference	Difference %	Significance
Q1	0.456	0.461	-0.005	-1.03%	ns
Q2	2.434	2.251	0.183	8.15%	**
Q3	1.505	1.445	0.060	4.14%	**
Q4	1.083	1.040	0.042	4.09%	ns
Q5	3.003	2.881	0.122	4.22%	*
Q6	0.324	0.332	-0.008	-2.32%	ns
Q7	2.441	2.345	0.096	4.09%	**
Q8	3.101	2.998	0.103	3.43%	*
Q9	3.739	3.701	0.038	1.02%	ns
Q10	1.521	1.513	0.008	0.52%	ns
Q11	1.272	1.237	0.035	2.84%	ns
Q12	1.044	1.023	0.021	2.03%	ns
Q13	0.734	0.717	0.017	2.36%	ns
Q14	0.817	0.801	0.016	1.97%	ns
Q15	0.841	0.823	0.019	2.26%	ns
Q16	1.184	1.151	0.034	2.93%	ns
Q17	1.815	6.201	-4.386	-70.73%	-
Q18	1.738	1.582	0.156	9.86%	**
Q19	1.142	1.064	0.078	7.28%	*
Q20	7.404	10.902	-3.499	-32.09%	-
Q21	3.129	3.160	-0.030	-0.96%	ns
Q22	0.889	0.862	0.028	3.20%	ns

In the next experiments we use a synthetic workload to investigate the impact of number of joins and number of inequality conditions on planning time. In this experiment we (i) vary the number of joins (equality or inequality) in a query from 0 to 25, and (ii) vary the number of conditions for a single join. The results are shown in Figure 5.1. Figure 5.1a shows the planning time for varying number of joins that are related using an equality condition, and we can see that the planning time increases substantially with an increasing number of joins. This is the result of the increasing number of choices posed to the optimizer (join orders and

algorithms).

In Figure 5.1a we can see that the number of joins has little impact on our RMJPATCH. Figure 5.1b shows a similar experiment, but this time instead of equality conditions the joins are related using inequality conditions, i.e., ($<$ instead of $=$). We can see that the optimizer requires much less time compared to the case of equality joins. In both cases we can see that RMJPATCH does not impose a large overhead on planning time for a larger number of joins compared to HEAD.



(a) Varying number of joins with equality condition

(b) Varying number of joins with inequality condition

(c) Varying number of equality conditions

(d) Varying number of inequality conditions

Figure 5.1.: Planning time for varying number of joins and conditions.

Figure 5.1c and Figure 5.1d compare the planning time for respectively, a varying number of equality and inequality conditions. In this case we do not vary the number of joins, i.e., the conditions relate to only one join. We can see that the planning time is very low as compared to varying the number of joins. RMJPATCH only incurs an overhead for inequality conditions due to the checking of potential range conditions. Also in this case the overhead is relatively small.

5.3.3. Execution Time

Next we show the results of the TPCH queries in terms of execution time. The results are shown in Table 5.4. Recall that we exclusively use sort-merge joins for the execution of these queries in order to quantify if our patch adds overhead to the traditional sort-merge join execution algorithm due to additional conditional statements on a Boolean flag for the RMJ

(the conditionals check if the execution is a RMJ or an MJ). We can see that there are queries with statistical significance, but by consulting the difference we can see that the RMJPATCH is faster most of the time compare to HEAD. Due to the large sample size (100) in most of the cases we will have some kind of statistical significance, but since none of these queries use the RMJ algorithm our patch cannot be faster for these queries, and the difference in runtime is most probably only caused by interference of other processes. In summary, we can conclude, that the overhead of our RMJPATCH to the traditional MJ execution algorithm is negligible.

Table 5.4.: Execution time in milliseconds for the TPCH queries

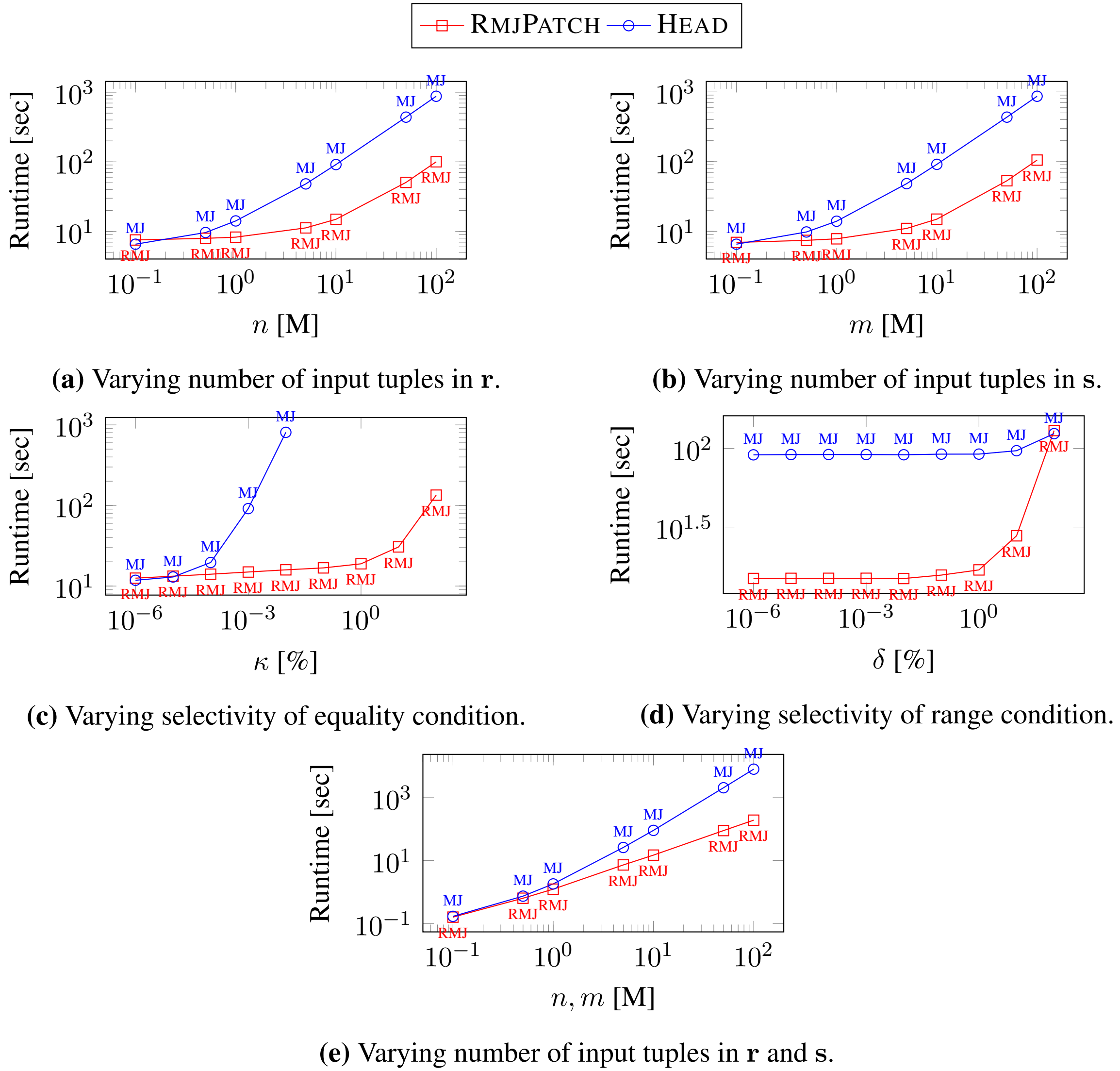
Query	RMJPATCH	HEAD	Difference	Difference %	Significance
Q1	20,901	20,956	-55	-0.26%	ns
Q2	4,098	4,097	1	0.02%	ns
Q3	17,364	17,464	-100	-0.57%	ns
Q4	2,548	2,567	-18	-0.71%	ns
Q5	20,179	20,270	-90	-0.45%	ns
Q6	3,303	3,386	-84	-2.47%	****
Q7	14,267	14,274	-7	-0.05%	ns
Q8	22,527	22,375	152	0.68%	ns
Q9	47,211	48,301	-1,090	-2.26%	****
Q10	10,789	10,852	-63	-0.58%	*
Q11	4,654	4,625	29	0.62%	ns
Q12	6,489	6,578	-88	-1.34%	ns
Q13	14,646	14,790	-145	-0.98%	****
Q14	4,191	4,260	-69	-1.62%	****
Q15	4,224	4,347	-123	-2.84%	****
Q16	5,100	5,112	-12	-0.23%	ns
Q17	3,299,008	3,333,433	-34,425	-1.03%	-
Q18	79,072	79,344	-271	-0.34%	ns
Q19	5,363	5,467	-104	-1.90%	****
Q20	7,056,775	7,308,710	-251,936	-3.45%	-
Q21	25,197	25,186	11	0.05%	ns
Q22	11,262	11,303	-41	-0.36%	ns

5.4. Range Join Execution Time

5.4.1. RMJ Vs. MJ

In the next experiment we compare the sort-merge join of PostgreSQL with our range merge join on the synthetic workloads (cf. Section 5.1.1). For these first experiments we disable the other hash and nested loop join algorithms. The results are shown in Figure 5.2. In the figures we also indicate for each approach the used join algorithm sort-merge join (MJ) that only supports equality conditions or range merge join (RMJ) that supports both equality and range conditions. As expected, the RMJ can take advantage of the range condition and outperforms the MJ. This is particularly visible in Figure 5.2d for cases when the range condition is very selective (small values of selectivity), and when the equality condition in Figure 5.2c has a higher selectivity, which means that the MJ produces more result tuples for which the range

condition has to be performed as a filter after the join. When δ reaches 100%, i.e., all tuples that satisfy the equality condition also satisfy the range condition, RMJ and MJ have the same performance in this case.



(a) Varying number of input tuples in **r**.

(b) Varying number of input tuples in **s**.

(c) Varying selectivity of equality condition.

(d) Varying selectivity of range condition.

(e) Varying number of input tuples in **r** and **s**.

Figure 5.2.: Runtime results for our synthetic workloads with only MJ and RMJ enabled.

5.4.2. Merge Condition Vs. Range Condition

In the next experiment, we analyze the efficiency of the merge and range condition in the execution of a sort-merge based join. We are interested to see which join condition, equality condition using MJ or range condition using RMJ is more efficient to evaluate based on their selectivity. In both cases we use a sort-merge based join (MJ or RMJ) and the same selectivity, but we either use an equality condition without range condition or a range condition without equality condition. The results are shown in Figure 5.3. As expected, for cases where we only

have a range condition (RANGE) a RMJ is used and for cases when we only have an equality condition (MERGE) a MJ is used. The sort-merge technique for the equality condition generally is more efficient than for the range condition, with an increased runtime of 7%, 14%, 18%, 18%, and 21% for the provided data points of respectively, 0.0001%, 0.001%, 0.01%, 0.1%, and 1% selectivity. The reason for this is the more efficient backtracking mechanism for equality that can exploit the transitivity property of equality (cf. line 12–13 in Algorithm 1). More specifically, if a new outer tuple does not match in equality with a marked inner tuple, then backtracking is not required, because no inner tuples between the marked tuple and the current inner tuples can produce a join match. For the range condition this does not hold and, particularly, in the absence of an equality condition backtracking always has to be performed, resulting in more and further backtracking (cf. line 12–13 in Algorithm 2). It is worth mentioning at this point that despite backtracking is less effective for the range condition, an inner tuple may only be skipped once, i.e., when it can no longer produce join matches. Thus, the complexity of the sort-merge techniques for both equality condition and range condition is the same.

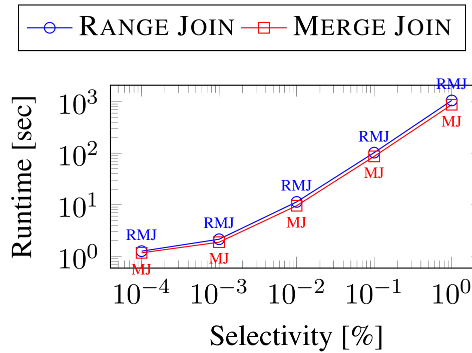
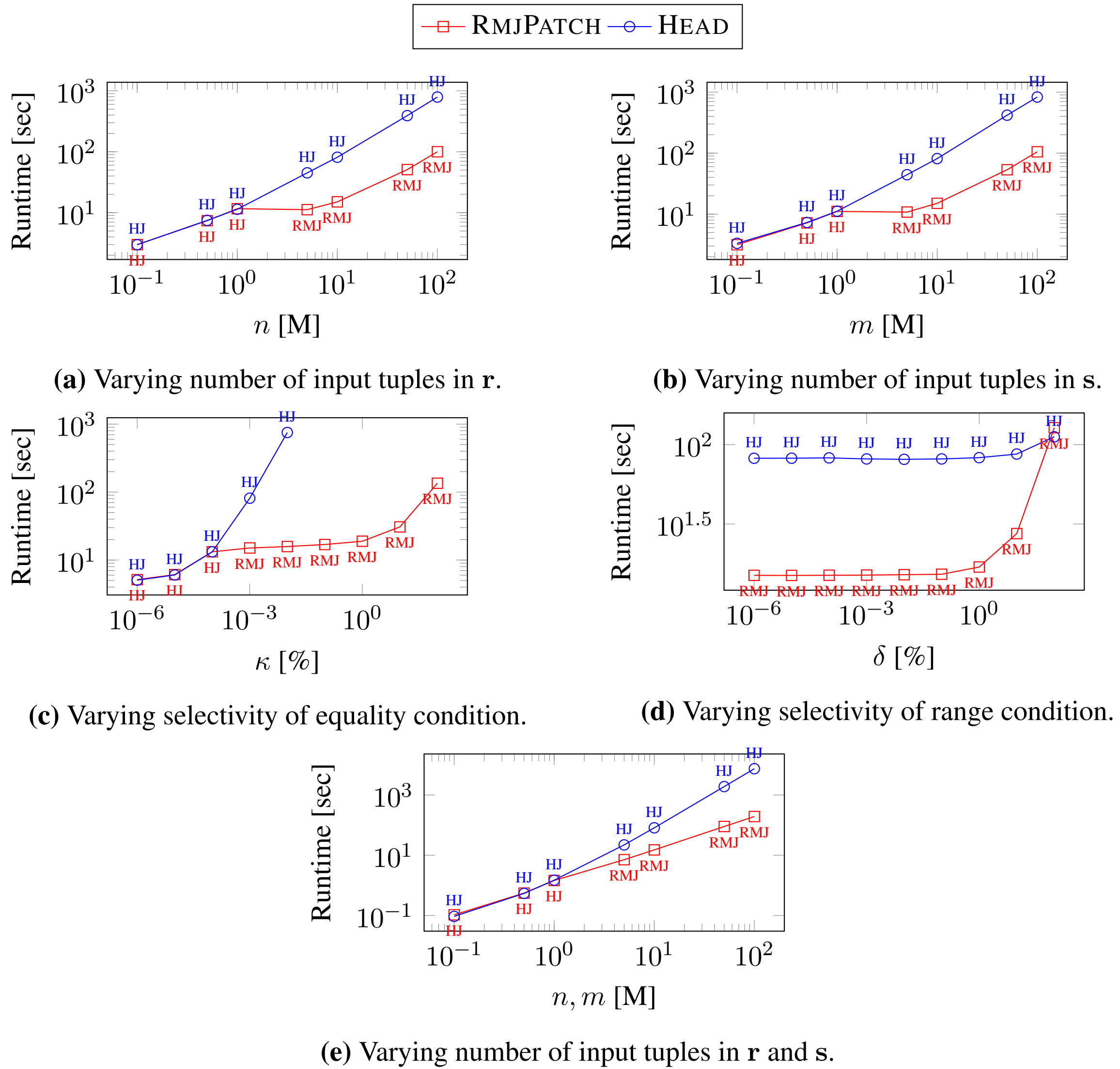


Figure 5.3.: Runtime results for varying selectivity κ or δ.

5.4.3. RMJPATCH VS. HEAD

In this part of the evaluation we enable all join algorithms and compare RMJPATCH with HEAD on our synthetic workloads. The results are shown in Figure 5.4. We can see that in general PostgreSQL favours the hash join (HJ) to the sort-merge join. In fact for HEAD the optimizer never chooses the MJ, and comparing to Figure 5.2 where the MJ is slightly slower compared to the HJ the optimizer opts for the correct choice. We can see that also for RMJPATCH the optimizer chooses a HJ in a few settings, in particular when one or both relations are small so that efficient in-memory hash tables can be employed and when the equality condition is very selective. We can see for RMJPATCH that in all cases when the RMJ is faster than the HJ, the optimizer correctly opts for it, resulting in a more efficient join execution.

Next, we compare RMJPATCH and HEAD on our synthetic workloads but this time with range condition only, i.e., without equality condition. For this cases, since HEAD has no adequate join algorithm we have to reduce the default values for the number of input tuples in $\mathbf{r}$ and $\mathbf{s}$ to $10,000$. The results are shown in Figure 5.5. In this case the only join option for



(a) Varying number of input tuples in $\mathbf{r}$.

(b) Varying number of input tuples in $\mathbf{s}$.

(c) Varying selectivity of equality condition.

(d) Varying selectivity of range condition.

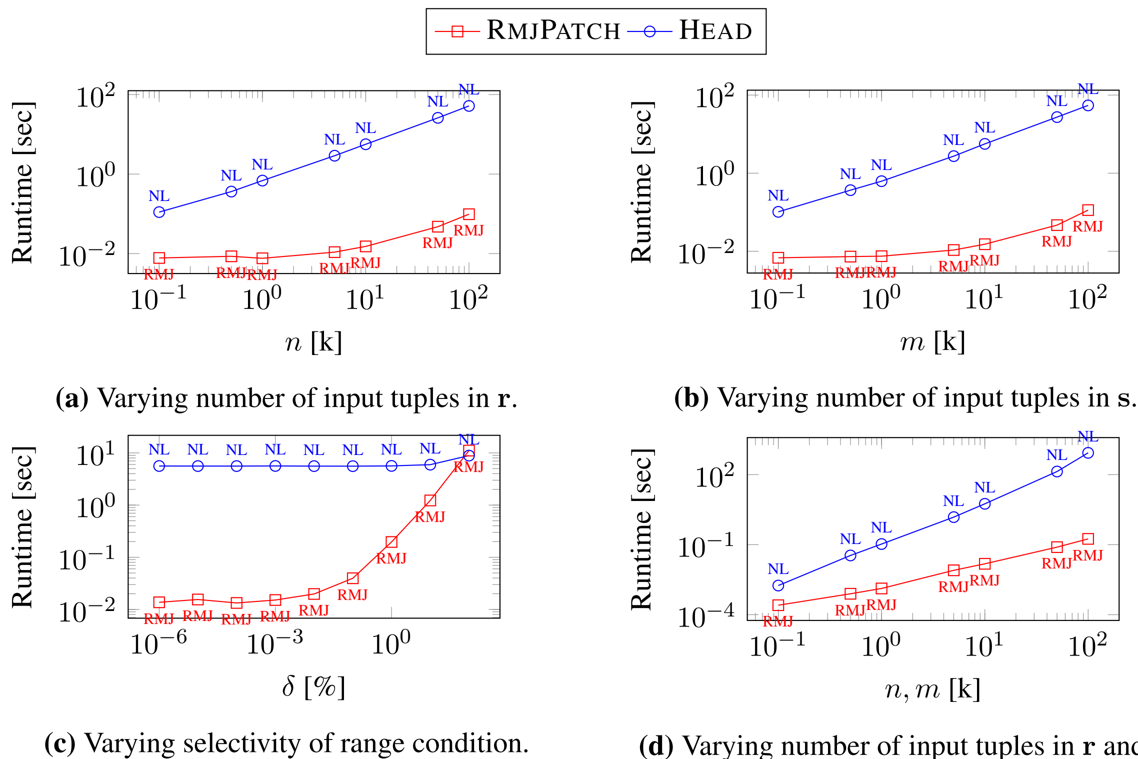
(e) Varying number of input tuples in $\mathbf{r}$ and $\mathbf{s}$.

Figure 5.4.: Runtime results for our synthetic workloads with all join algorithms enabled.

HEAD is a nested loop (NL), while RMJPATCH can take advantage of the range condition and use a RMJ and is much faster. When the selectivity of the range condition reaches 100%, i.e., all tuples match on the range condition, both NL and RMJ have the same performance. In this case, the NL spends more time for checking the range condition compared to RMJ, but RMJ additionally has to sort the input resulting in this worst case scenario to the same execution time for both approaches.

5.4.4. RMJ Vs. Index Joins

PostgreSQL supports three types of indices that, if created upfront, can act as an access method for a range condition and thus, potentially can be used to compute a range join using an index



(a) Varying number of input tuples in **r**.
(b) Varying number of input tuples in s.
(c) Varying selectivity of range condition.
(d) Varying number of input tuples in **r** and s.

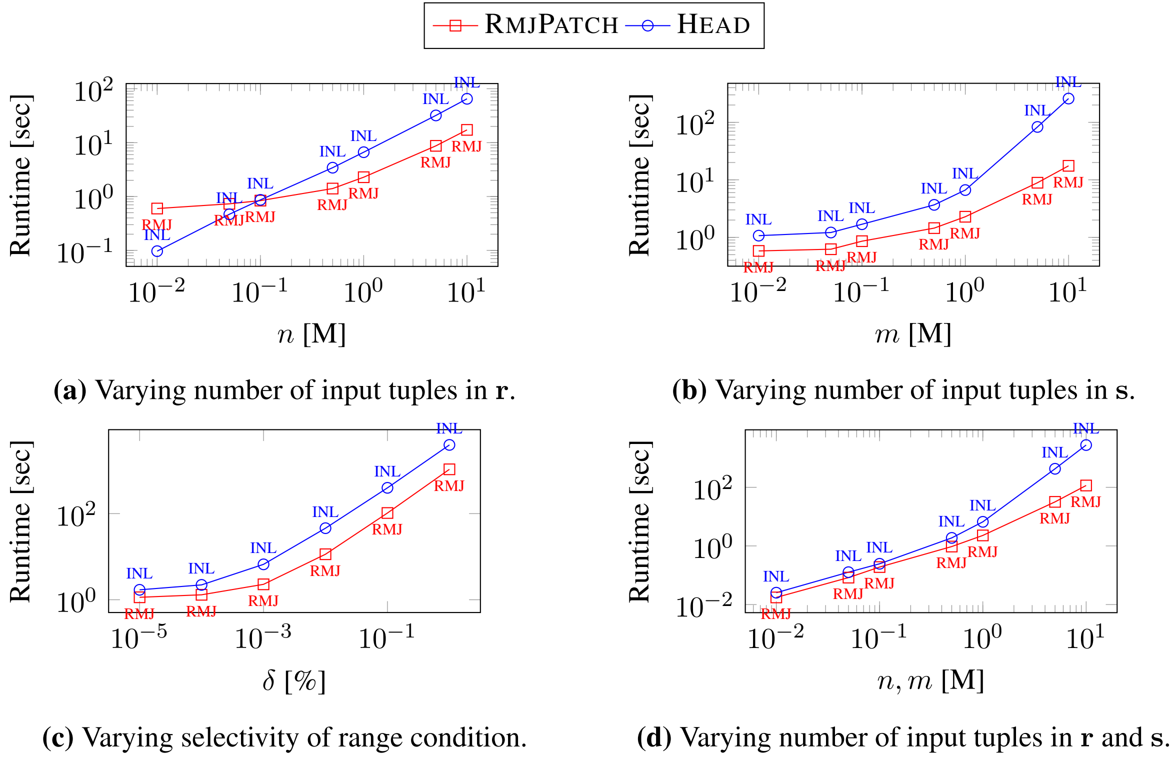
Figure 5.5.: Runtime results for our synthetic workload with range conditions only (smaller default values for input relations, $n = m = 10$k).

join. For PostgreSQL's range types[2] two index types exist, the general inverted search tree (GiST) [13] and the space partitioned general inverted search tree (SP-GiST) [2]. The GiST index type is an implementation of a one-dimensional R-Tree, while the SP-GiST index types is an implementation of a quadtree. Both of these indices can be created on range types and provide an access method for the operators `@>` and `<@` that correspond to "range contains element" and "element is contained by range" respectively. The third index types that supports range conditions is the B+-Tree [16]. When created on a scalar value, such as for instance a `DATE` or an `INT`, it supports range searches over these values. It is important to note at this point that GiST and SP-GiST need to be created on the relation containing the range (denoted as **r** in this thesis), while the B+-Tree index needs to be created on the relation containing the value or element (denoted as s in this thesis).

Figure 5.6 compares Head using a B+-Tree index with our RMJ. We use a default of 1M tuples for our synthetic workloads in this case and only a range condition (no equality condition). As expected Head opts for an index nested loop or index join (INL) using the B+-Tree index. This type of join scans one (outer) relation (**r** in this case) and for each tuple performs a range search on the other (inner) relation (s in this case). RMJ is more efficient that INL in general, while INL is efficient if the number of outer tuples (n) is small. Only when the number of outer tuples (cf. Figure 5.6a) is smaller that ten times the inner relation (s) INL

[2]https://www.postgresql.org/docs/12/rangetypes.html

is more efficient than RMJ. This because RMJ scans both relations, while INL uses the index for the other, which in this case is the larger relation.



(a) Varying number of input tuples in $\mathbf{r}$.

(b) Varying number of input tuples in $\mathbf{s}$.

(c) Varying selectivity of range condition.

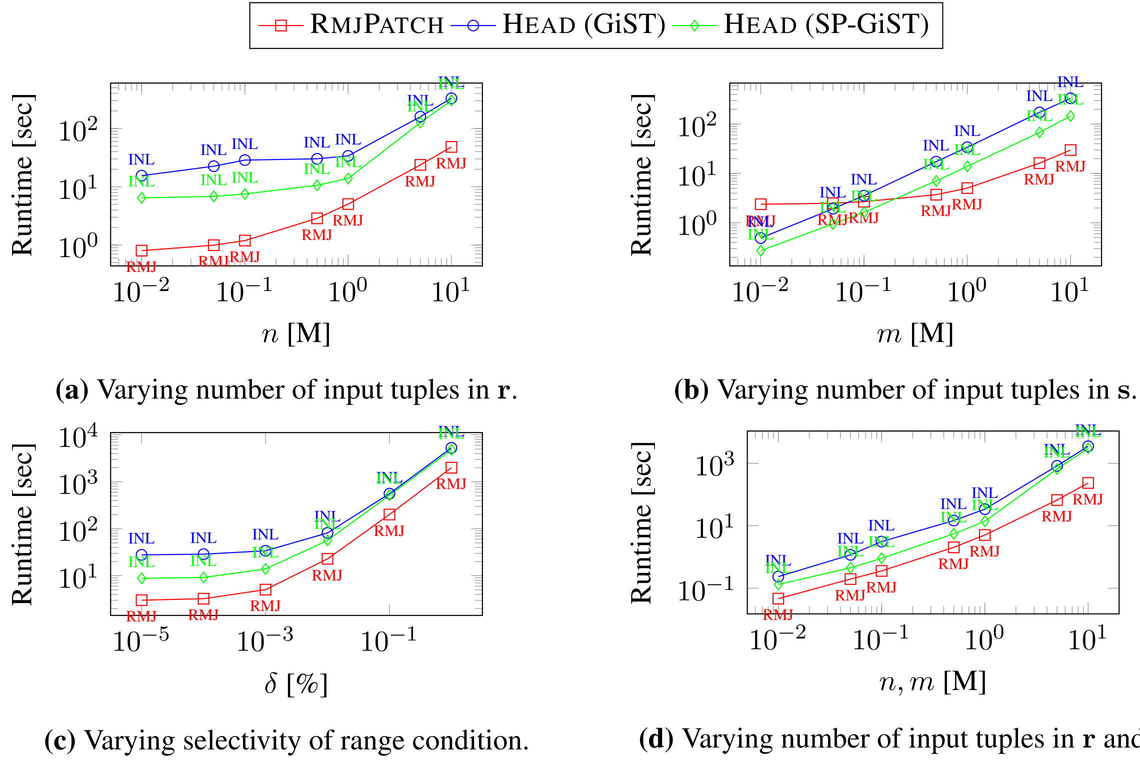
(d) Varying number of input tuples in $\mathbf{r}$ and $\mathbf{s}$.

Figure 5.6.: Runtime results for our synthetic workload with range conditions only and a B+-Tree index on s for HEAD (smaller default values for input relations, $n = m = 1$M).

Figure 5.7 shows the same experiment for range types where for HEAD a GiST or SP-GiST index has been created. We can see that RMJ in most settings is more efficient than the INLs based on GiST and SP-GiST, except when the relation s contains fewer tuples. Recall that for this case the indices are created on $\mathbf{r}$ so for the INL $\mathbf{r}$ is the inner relation and s is the outer relation (opposite of B+-Tree). Similar as in the previous experiment INL is only efficient when the outer relation contains about ten times less tuples then the inner relation (cf. Figure 5.7b).

5.4.5. Real-World Workloads

Next, we evaluate RMJPATCH and compare it to HEAD on our real-world workloads as described in Section 5.1.2. The results are shown in Figure 5.8. The labels for the workload are indicates with the dataset and the used equality attribute. For instance, "Incumbents (ssn)" is the Incumbents dataset with equality condition on the ssn attribute, while "Incumbents ()" is the Incumbents dataset with no equality condition. Additionally, also here we provide the execution algorithm used for RMJPATCH and HEAD, and we also report the numbers for RMJPATCH (RMJ only) where we force the optimizer to use our range merge left outer



(a) Varying number of input tuples in **r**.

(b) Varying number of input tuples in **s**.

(c) Varying selectivity of range condition.

(d) Varying number of input tuples in **r** and **s**.

Figure 5.7.: Runtime results for our synthetic workload with range conditions only and a GiST or SP-GiST index on **r** for HEAD (smaller default values for input relations, $n = m = 1$M).

join (RMLJ). For "Incumbents (ssn)", both RMJPATCH and HEAD opt for a hash right outer join (HRJ) with inverted inputs that is the fastest option for this setting. We can see that if we force the optimizer to use a RMLJ the performance is very similar 255ms compared to the 230ms of the HRJ. The reason why the hash join approach is extremely efficient for this query is because the ssn attribute, with a selectivity of 0.0027%, is very selective, while the range condition only provides an additional selectivity of 28%. For "Incumbents (pcn)", RMJPATCH opts for a RMLJ while HEAD opts for a sorted-merge left outer join (MLJ). Also in this case the performance of MLJ and RMLJ is very similar, with 320ms for the RMLJ and 355ms for the MLJ, because also in this case attribute pcn is very selective (0.01%) and the range condition provides an additional selectivity of 17%. For "Incumbents ()" there is no equality condition anymore and HEAD has to revert to a NL, while RMJPATCH can use the range condition and use a RMLJ, resulting in a huge performance improvement of 1.7sec compared to 11.5sec. The selectivity of the range condition in this case is 6.5%.

For "Flight (fap)" RMJPATCH used a RMLJ compared to a MLJ used by HEAD. The RMLJ takes 600ms in this case while the MLJ takes 7.6sec. The selectivity of attribute fap in this case is 1.3% and the selectivity of the range condition on top of this is 1.6% that is much more selective as in the previous cases. For "Flight ()" with no equality condition HEAD has to use a NL that takes 30sec, while RMJPATCH can use a RMLJ that takes 1sec with a range condition having a selectivity of 1.2%.

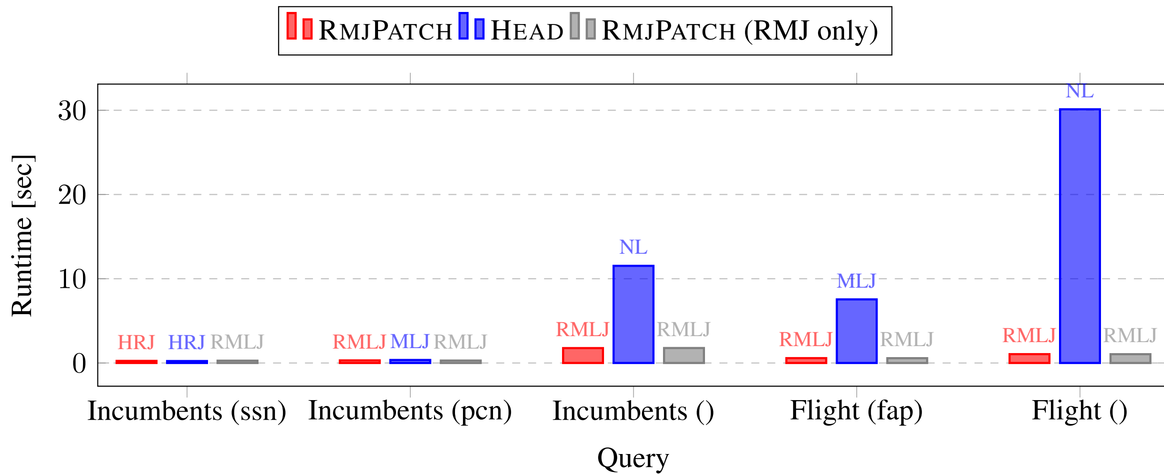


Figure 5.8.: RmjPatch vs. Head on real-world workload.

5.5. Summary

In summary, our RmjPatch and its implementation of the RMJ does not cause large overheads in terms of planning time, and its additions to the traditional MJ (checking of additional flags during execution) does not have an effect on the MJ's performance. For cases when queries contain range joins (with or without equality conditions) the RMJ provides considerable performance gains over a large spectrum of settings as well as real-world datasets. The integration of the RMJ in RmjPatch also shows that the planner is able to detect cases when the RMJ is adequate for query processing.

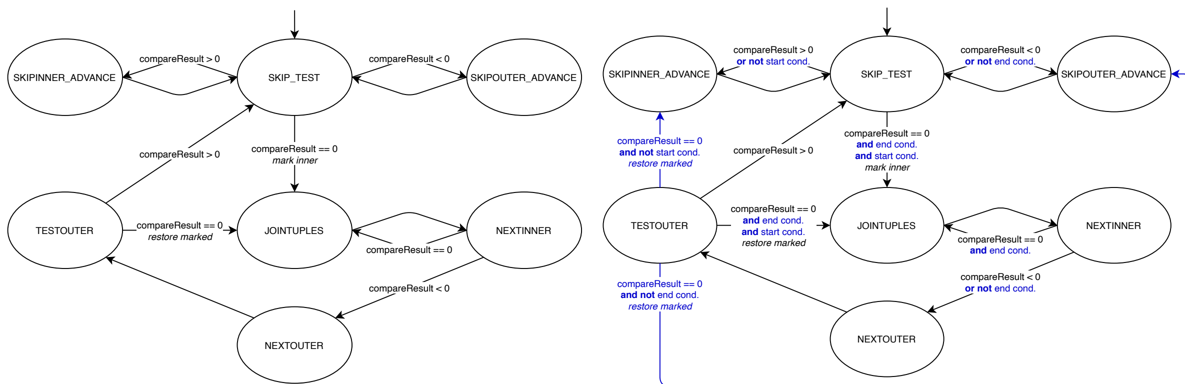
6. Experiences from the PostgreSQL Implementation

In this section we share our experiences in the PostgreSQL development and the initial steps, approach, and methodology that was used through the development of this thesis.

6.1. Procedure

When starting the work, the first step was to get a deep understand of PostgreSQL's implementation of the Merge Join. We created a simplified state diagram shown in Figure 6.1a. To test our initial modifications, we implemented this simple version of the MJ algorithm as a small C program to have a prototype for quick implementation. We then modified the simplified state diagram with the additional conditions for the Range Merge Join, and verifying our theoretical solutions by modifying the implemented algorithm accordingly.

As a simple starting point, we worked on a solution using an extra state we called `RANGE_TEST` which would sit upstream of `JOINTUPLES` checking the range conditions every time before tuples get joined. This was effectively the same as the normal MJ implementation. To get the advantages of a Range Join we then evaluated the start condition in the states which are upstream and only checked the end condition in `RANGE_TEST`. This meant, it made no sense anymore to have an extra state for that. Our second attempt for a RMJ implementation, shown in Figure 6.1b, was the one which we ended up implementing into PostgreSQL.



(a) Vanilla merge join.

(b) Modified version including range conditions.

Figure 6.1.: Simplified state diagrams of the PostgreSQL Merge Join state machine.

We started by using only range conditions consisting of two inequality expressions and did not support range types. We did not implement sorting at this early stage yet, and instead sorted **r**.ts and **s**.t in the query itself using `SELECT s ORDER BY s.t` and `SELECT r ORDER BY r.ts` as a sub-query. To compare the range attributes, we just had to initialize both conditions as executable expressions and execute them in our comparison function.

The next step, was to identify range conditions consisting of a single containment expression for range types. To execute the RMJ with this type of range condition, we had to find a new solution for our comparison. We tried different solutions, including extracting the specific values from the current tuples, until we found out, that we can initialize the arguments separately as executable expressions, to use them with the internal range type functions. This seemed the cleanest solution to us, since we used PostgreSQL's abstractions and implemented two new range type functions to check for start- and end condition.

At this point, we create only one path using the first identified range condition to have a running prototype. To identify and try out multiple range conditions, we defined a range condition as a list containing either two inequality expressions or one containment expression. The planner gets a list containing all possible range conditions, i.e. a list containing lists, and a path has a single range condition, which is a list itself. To give the planner the option to choose the best range condition, we had to implement sorting.

The sorting was the most difficult part, because we needed to modify multiple parts of the planning process. We identified the point, where the pathkeys are created and ordered, but we needed equivalence classes to create pathkeys. So we wanted to create equivalence classes first, but to create them, we need the btree operator families.

For the inequality expressions we could just look up the operator family of the used inequality operator like it is done for equality expressions, because the same are also used for sorting both sides. For the containment expressions we had to find another solution. Both sides of the operation have a different typeand these are not even clearly defined by the operator, so we had to work with typecache lookups and use the cached btree opamilies.

Having the correct opfamilies, we needed to create the single-member equivalence classes. But if we used the types implied by the operator, as it is done for merge clauses, the element side would not have the correct type for the btree opfamily. So we extracted the types directly from the arguments, but then the range side would not have the correct type. We have to check, which one is the element type by checking if it is a "contains" or "contained by" operation and only extract this side separately.

6.2. Takeaways

To us it was very helpful to visualize the state machine as a state diagram. We were able to visualize and simulate our solutions without any coding by simply modifying the diagram and traversing it with different examples. Then we performed the more sophisticated tests by using our simple standalone RMJ state machine algorithm. In this way, we did not have to care about any PostgreSQL-specific implementation to test our initial solution.

The code-base of PostgreSQL is very large as well as constantly evolving. There are multiple levels of abstraction and not everything is intuitive as it seems at a first glance, although the

comments in the code are extremely helpful. Every time we had to implement something new, we searched for a similar implementation in the existing code-base to have some reference. We derived most of the methods used for handling the range clauses from existing methods used for merge clauses. These gave a good framework to adapt to our needs, especially with the inequality expressions.

It was very helpful to us to reflect on the work we did, and we cleaned up a lot of the code while writing this thesis, in particular Chapter 4. Eventually, we even found some bugs that were not showing in any of our tests.

Working with range types and containment expressions was much more complex, because we could not derive it from the merge clause handling. We searched for different examples in the code, we had multiple iterations of different solutions. Some solutions still feel like a workaround and can possibly be implemented cleaner. We have to work with multiple typecache lookups, we compare the expressions operator number (`opno`) with the specific operator numbers referenced by `OID_RANGE_CONTAINS_ELEM_OP` and `OID_RANGE_ELEM_CONTAINED_OP` multiple times and we introduced two new rangetype specific functions, but only internal.

The most difficult part, was to get the sorting to work. Multiple times, we reached a position in the code, where we thought the sorting implementation would start, just to realize that some information needs to be assigned further upstream. The overhead for every joinable inequality expression shown in Figure 5.1 is a result of pushing the creation of equivalence classes further upstream. This would not be necessary in the current state of the implementation, but it is a result of a first try to implement the RMJ working with presorted paths. We postponed this, because we would need to reintroduce the distinction between outer and inner paths, which is not necessary for the traditional MJ.

6.3. Open Issues

Consider Presorted Sub-Paths. As mentioned in the previous section, we already started with the implementation for the RMJ working with presorted paths. At first the optimizer creates the merge join paths that require an explicit sorting of the relations, which is the procedure where we step in to create RMJ paths. Then the optimizer tries to match different outer sub-paths in their respective sort order with subsets of all merge clauses. The goal is to find a combination of a presorted outer path and the corresponding set of merge clauses, which is more efficient then the explicitly sorted paths. We want to modify this procedure to also take range clauses into account. Additionally, the merge join does not require to do the same for inner paths, because there is no difference between outer and inner. The RMJ on the other hand needs this distinction, which provides an additional challenge.

RMJ-Specific Test Cases. PostgreSQL has an integrated test suite. For our RMJ implementation, we will need specialized tests to ensure its integrity during future development.

Parallelization. In the current state, parallelization is disabled for our RMJ, while it is enabled for the traditional MJ. We want to make it possible to use partial merge joins with

range conditions.

Path Pruning. Section 4.3.2 described how many paths are created, if we have merge- and range conditions. This number grows exponentially with the number of merge clauses and potential range clauses. Our goal is to reduce this number of paths in a way that prevents the loss of the most efficient paths in most cases.

Quickly Produce Lower-Bound Cost Estimates. For the pruning of unpromising join paths at an early stage PostgreSQL relies on lower-bound cost estimates. When the MJ or RMJ contain merge clauses PostgreSQL uses only the first merge clause to provide estimates. For cases when no merge clause exists, currently we do not provide initial costs in the function `initial_cost_mergejoin`, but rather ignore the range clauses. Related to this, we also want to rethink how the final costs are calculated. Based on the experiments shown in the Sections 5.4 and A.3, it seems that the optimizer does not always choose the best possible path. We want to evaluate that behaviour further and provide an improved solution.

7. Conclusion and Future Work

In this thesis, we formally define range joins and provide an implementation based on the traditional sort-merge paradigm. We then, described in detail the implementation of PostgreSQL's sort-merge join as a state machine and show how it can be extended to be able to additionally handle range joins. More precisely, we modified three states and the corresponding transitions and explain how these modifications enable PostgreSQL to execute a range merge join.

We provide a tight integration of our range merge jon into the PostgreSQL kernel, adhering to the PostgreSQL architecture and conventions. The implementation into PostgreSQL's code-base is described in detail, split into the four parts: definition of range clauses, sorting, initialization, and execution. In our experiments, we showed how the RMJ incurs only a very small overhead in planning time for specific applications, and no significant overhead in execution time for the traditional sort-merge join caused by our extension. Using synthetic workload, we showed how the RMJ outperforms the sort-merge join, as well as the hash join in most settings when a range condition in addition to an equality condition is present. Compared to index joins that can support range joins, we were able to reduce the execution time significantly. Our experiments on real-world workloads reveal how our implementation has the ability to make an important difference in practical applications. Finally, we also provided an insight into our experience with the integration of new functionality into PostgreSQL together with the insights we gained during this work.

Future work points in several directions. First, we want to submit the patch to the psql-hackers[1] mailing list, to gain feedback from the community, which will be helpful with our future development. Then, we want to work on the open issues described in the thesis. In particular, we want to implement the RMJ with presorted sub-paths, we want to expand the test suite with RMJ-sepecific test cases, we want to develop a parallelized version of the RMJ, we want to optimize the planning by creating less paths, and we want to better integrate the cost estimates for range conditions into the optimizer.

[1] `https://www.postgresql.org/list/pgsql-hackers/`

Bibliography

[1] TPC-H specification – Transaction Performance Council. `http://www.tpc.org`. Accessed: 2020.

[2] W. G. Aref and I. F. Ilyas. SP-GiST: An extensible database index for supporting space partitioning trees. *J. Intell. Inf. Syst.*, 17(2-3):215–240, 2001.

[3] A. Behrend and G. Schüller. A case study in optimizing continuous queries using the magic update technique. In *Proceedings of the Conference on Scientific and Statistical Database Management, SSDBM '14*, pages 31:1–31:4. ACM, 2014.

[4] M. H. Böhlen, A. Dignös, J. Gamper, and C. S. Jensen. Database technology for processing temporal data (invited paper). In *Proceedings of the 25th International Symposium on Temporal Representation and Reasoning, TIME 2018*, volume 120 of *LIPIcs*, pages 2:1–2:7. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[5] M. H. Böhlen, A. Dignös, J. Gamper, and C. S. Jensen. Temporal data management - an overview. In *Business Intelligence and Big Data - 7th European Summer School, eBISS 2017, Tutorial Lectures*, volume 324 of *Lecture Notes in Business Information Processing*, pages 51–83. Springer, 2018.

[6] A. Dignös, M. H. Böhlen, and J. Gamper. Temporal alignment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012*, pages 433–444. ACM, 2012.

[7] A. Dignös, M. H. Böhlen, J. Gamper, and C. S. Jensen. Extending the kernel of a relational DBMS with comprehensive support for sequenced temporal queries. *ACM Trans. Database Syst.*, 41(4):26:1–26:46, 2016.

[8] A. Fuller. Look up values in sql server using range joins. `https://www.techrepublic.com/article/look-up-values-in-sql-server-using-range-joins/`, 2006. Accessed: 2020.

[9] A. Fuller. Using sql server joins for easy range lookups. `https://www.techrepublic.com/article/using-sql-server-joins-for-easy-range-lookups/`, 2006. Accessed: 2020.

[10] J. Gamper, M. H. Böhlen, and C. S. Jensen. Temporal aggregation. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2018.

[11] J. A. G. Gendrano, R. Shah, R. T. Snodgrass, and J. Yang. University information system (uis) dataset. TimeCenter CD-1, 1998.

[12] IP2Location.com. Ip address ranges by country. https://lite.ip2location.com/ip-address-ranges-by-country. Accessed: 2020.

[13] M. Kornacker. *Access Methods for Next-generation Database Systems*. PhD thesis, University of California, Berkeley, 2000. AAI9994590.

[14] E. Pitoura. Pipelining. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2018.

[15] PostgreSQL Global Development Group. Dcoumentation PostgreSQL 12 – Executor. https://www.postgresql.org/docs/12/executor.html, 2020. Accessed: 2020.

[16] D. Zhang, K. P. Baclawski, and V. J. Tsotras. B+-tree. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2018.

[17] J. Zhou. Hash join. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2018.

[18] J. Zhou. Index join. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2018.

[19] J. Zhou. Sort-merge join. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2018.

A. Appendix

A.1. TPC-H Query Description

Table A.1.: Description of TPC-H queries.

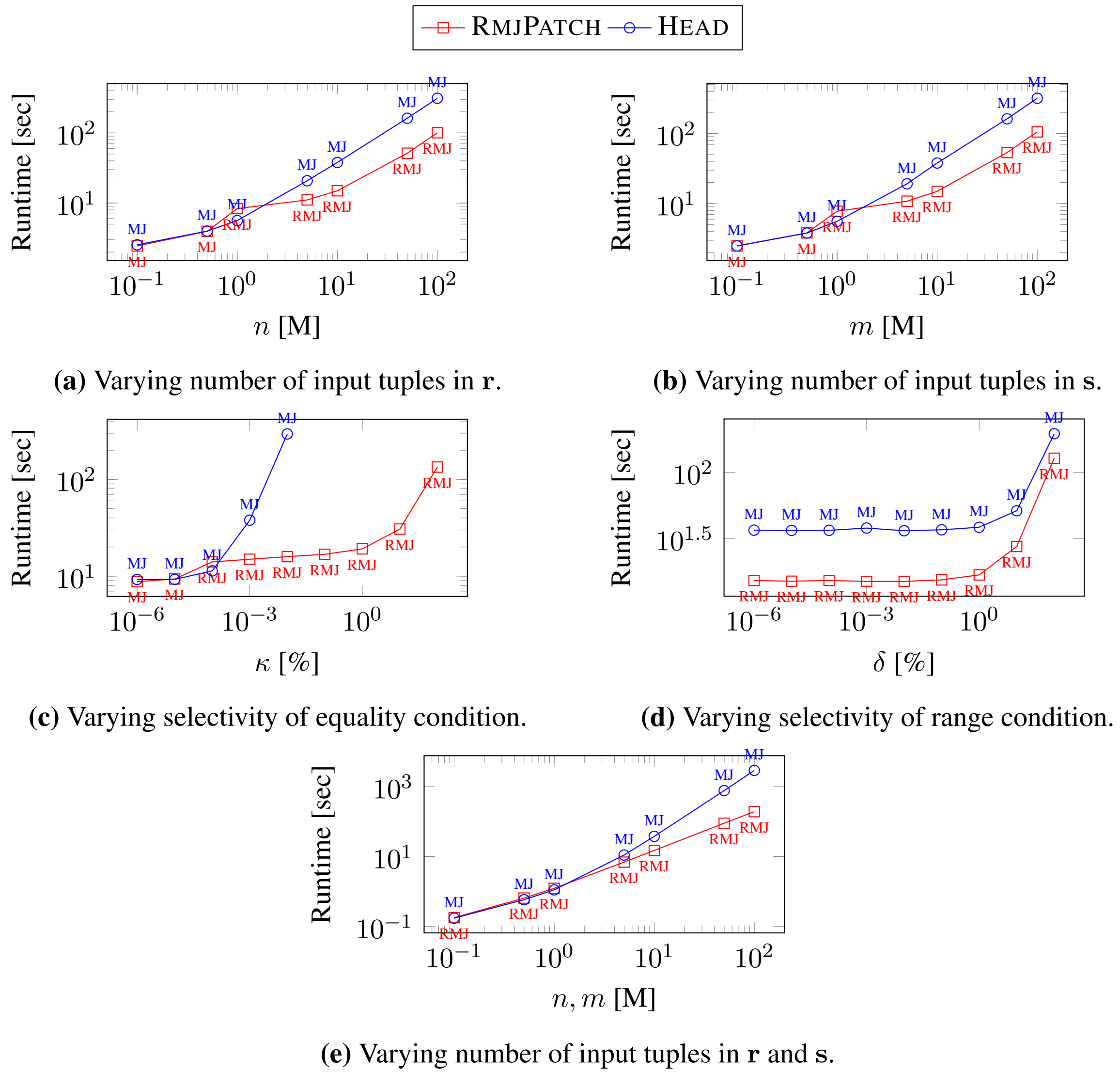
Query	Description	Operators
Q1	Pricing Summary Report	Aggregation (no join)
Q2	Minimum Cost Supplier	Join between five relations and a nested aggregation with join between four relations
Q3	Shipping Priority	Aggregation with join between three relations
Q4	Order Priority Checking	Correlated nested subquery (no explicit join)
Q5	Local Supplier Volume	Join between six relations
Q6	Forecasting Revenue Change	Aggregation (no join)
Q7	Volume Shipping	Aggregation with join between six relations
Q8	National Market Share	Aggregation with join between eight relations
Q9	Product Type Profit Measure	Aggregation with join between six relations
Q10	Returned Item Reporting	Aggregation with join between four relations
Q11	Important Stock Identification	Aggregation with join between three relations and a nested aggregation with join between three relations
Q12	Shipping Modes and Order Priority	Aggregation with join between two relations
Q13	Customer Distribution	Double aggregation with left join between two relations
Q14	Promotion Effect	Aggregation with join between two relations
Q15	Top Supplier	Aggregation with join between two relations and a nested aggregation
Q16	Parts/Supplier Relationship	Aggregation with join between two relations and a correlated nested subquery
Q17	Small-Quantity-Order Revenue	Aggregation with join between two relations and a correlated nested subquery
Q18	Large Volume Customer	Aggregation with join between three relations and a correlated nested subquery
Q19	Discounted Revenue	Aggregation with join between two relations and disjunctive predicates
Q20	Potential Part Promotion	Join between two relations and two correlated nested subqueries
Q21	Suppliers Who Kept Orders Waiting	Aggregation with join between four relations and two correlated nested subqueries
Q22	Global Sales Opportunity	Aggregation with two correlated nested subqueries (one with two levels of nesting)

A.2. TPC-H Planning Time Without Filtering Constants

Table A.2.: Planning time in milliseconds for the TPCH queries (without filtering constants).

Query	RmjPatch	Head	Difference	Difference %	Significance
Q1	0.916	0.918	-0.002	-0.20%	ns
Q2	4.147	4.115	0.032	0.78%	ns
Q3	2.726	2.650	0.076	2.85%	ns
Q4	1.961	1.897	0.064	3.35%	ns
Q5	5.215	5.115	0.100	1.96%	ns
Q6	0.721	0.617	0.104	16.85%	****
Q7	4.184	4.084	0.100	2.44%	ns
Q8	5.210	4.881	0.329	6.74%	*
Q9	6.158	6.350	-0.192	-3.03%	ns
Q10	3.018	2.951	0.067	2.27%	ns
Q11	2.170	2.129	0.041	1.93%	ns
Q12	1.930	1.905	0.025	1.29%	ns
Q13	1.758	1.688	0.070	4.15%	***
Q14	1.736	1.633	0.103	6.30%	***
Q15	1.495	1.445	0.050	3.45%	ns
Q16	2.200	2.187	0.013	0.58%	ns
Q17	1.833	1.423	0.411	28.86%	-
Q18	2.637	2.659	-0.022	-0.83%	ns
Q19	1.824	1.764	0.059	3.36%	ns
Q20	2.246	2.578	-0.332	-12.88%	-
Q21	5.179	5.175	0.004	0.08%	ns
Q22	1.478	1.472	0.006	0.42%	ns

A.3. Synthetic Experiments With Parallelization Enabled



(a) Varying number of input tuples in **r**.

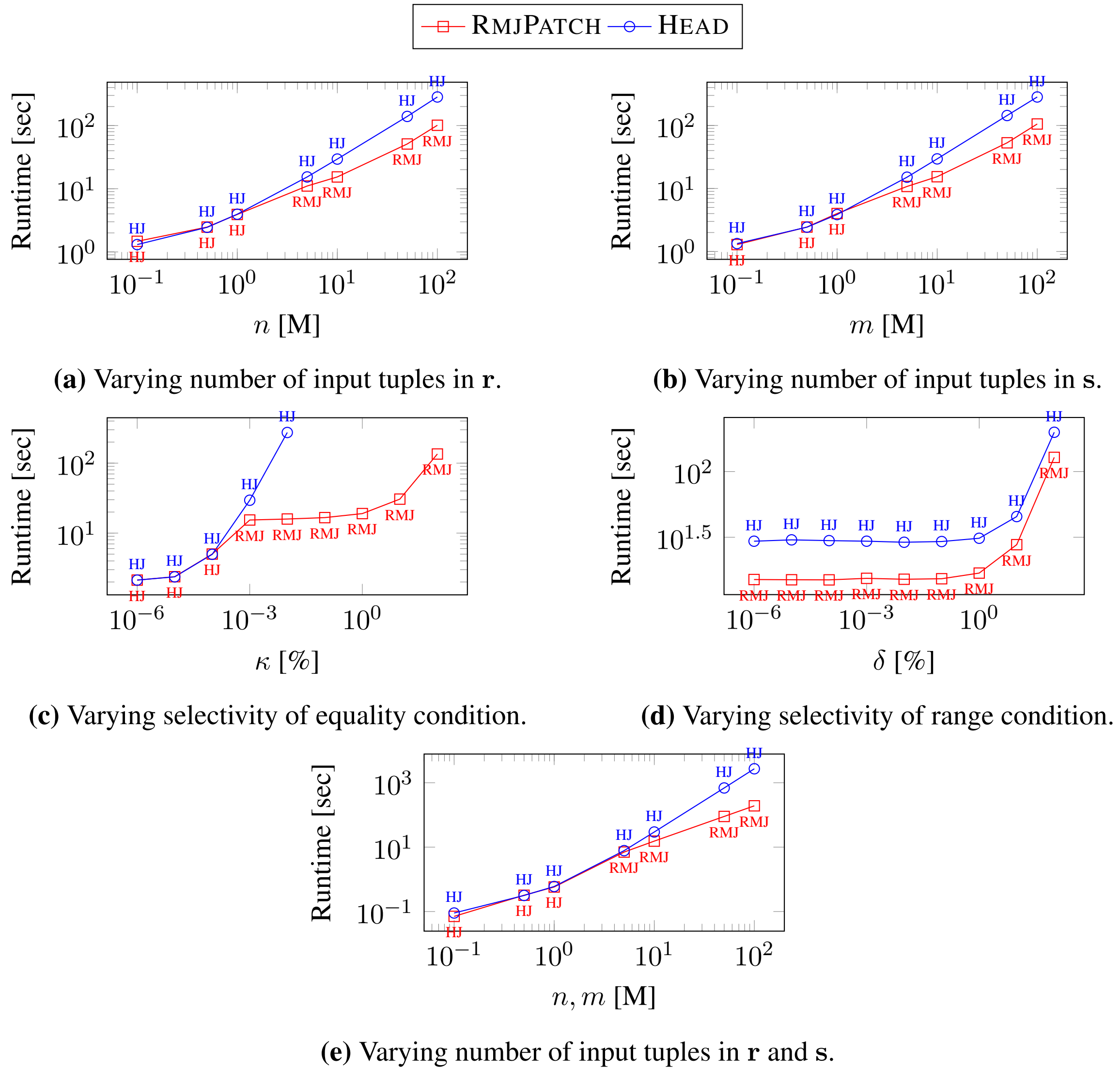
(b) Varying number of input tuples in **s**.

(c) Varying selectivity of equality condition.

(d) Varying selectivity of range condition.

(e) Varying number of input tuples in **r** and **s**.

Figure A.1.: Runtime results for our synthetic workloads with only MJ and RMJ enabled.



(a) Varying number of input tuples in **r**.

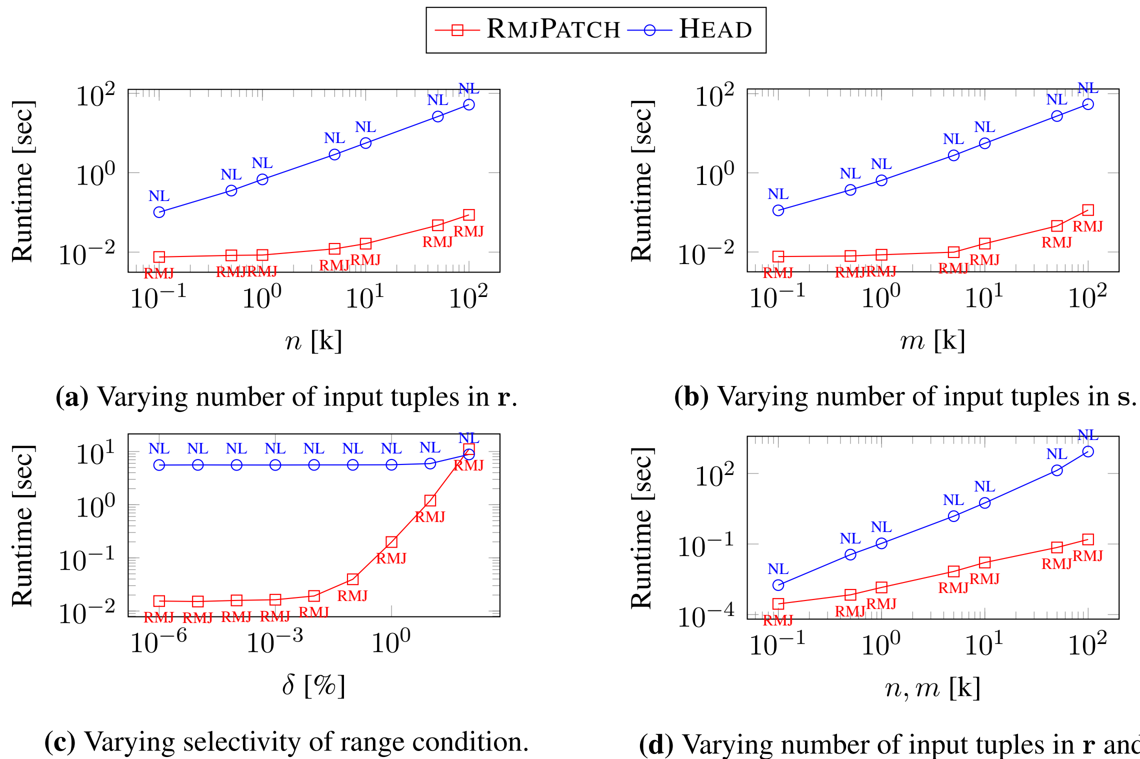
(b) Varying number of input tuples in **s**.

(c) Varying selectivity of equality condition.

(d) Varying selectivity of range condition.

(e) Varying number of input tuples in **r** and **s**.

Figure A.2.: Runtime results for our synthetic workloads with all join algorithms enabled.



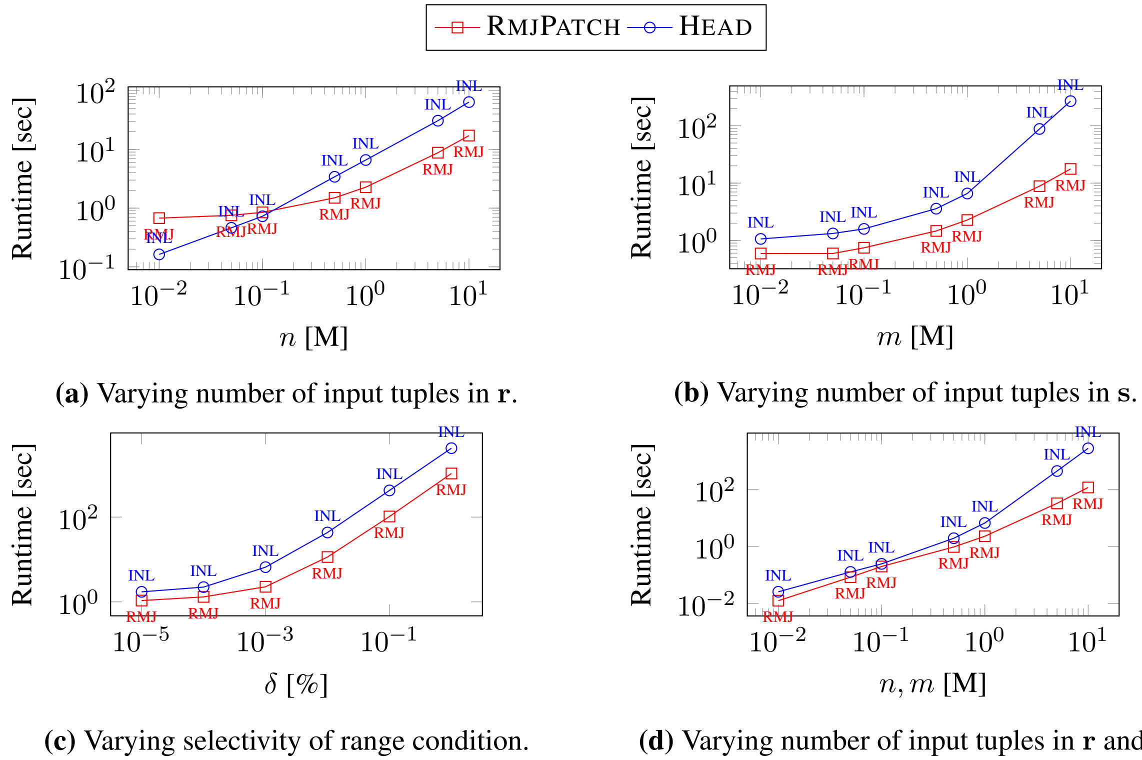
(a) Varying number of input tuples in $\mathbf{r}$.

(b) Varying number of input tuples in $\mathbf{s}$.

(c) Varying selectivity of range condition.

(d) Varying number of input tuples in $\mathbf{r}$ and $\mathbf{s}$.

Figure A.3.: Runtime results for our synthetic workload with range conditions only (smaller default values for input relations, $n = m = 10$k).



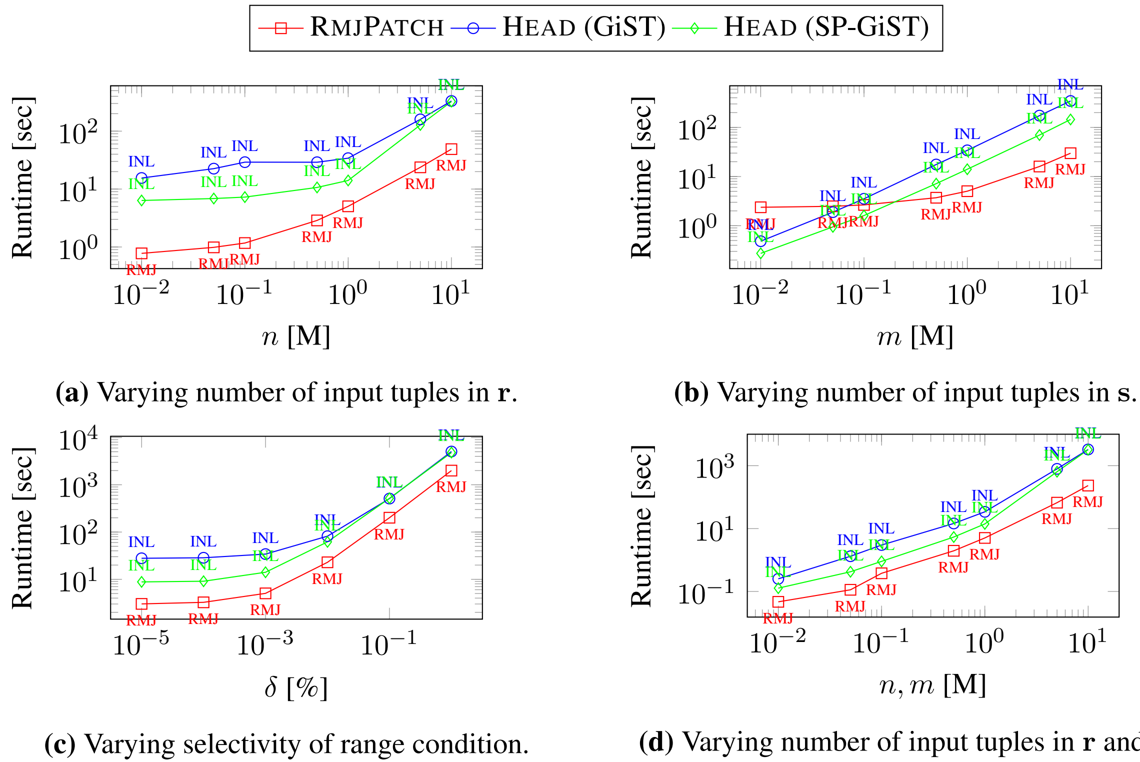
(a) Varying number of input tuples in **r**.

(b) Varying number of input tuples in s.

(c) Varying selectivity of range condition.

(d) Varying number of input tuples in **r** and s.

Figure A.4.: Runtime results for our synthetic workload with range conditions only and a B+-Tree index on s for HEAD (smaller default values for input relations, $n = m = 1$M).



(a) Varying number of input tuples in **r**.

(b) Varying number of input tuples in s.

(c) Varying selectivity of range condition.

(d) Varying number of input tuples in **r** and s.

Figure A.5.: Runtime results for our synthetic workload with range conditions only and a GiST or SP-GiST index on **r** for HEAD (smaller default values for input relations, $n = m = 1$M).