THE UNIVERSITY *of* EDINBURGH

A Unified Architecture for Efficient Binary and Worst-Case Optimal Join Processing

Amirali Kaboli

Master of Science by Research

THE UNIVERSITY OF EDINBURGH

2024

Abstract

Join processing is a fundamental operation in database management systems; however, traditional join algorithms often encounter efficiency challenges when dealing with complex queries that produce intermediate results much larger than the final query output. The emergence of worst-case optimal join (WCOJ) algorithms represents a significant advancement, offering asymptotically better performance by avoiding the enumeration of potentially exploding intermediate results.

In this thesis, we propose a unified architecture that efficiently supports both traditional binary joins and WCOJ processing. As opposed to the state-of-the-art, which only focuses on either hash-based or sort-based join implementations, our system accommodates both physical implementations of binary joins and WCOJ algorithms. Experimental evaluations demonstrate that our system achieves performance gains of up to $3.1\times$ (on average $1.5\times$) and $4.8\times$ (on average $1.4\times$) over the state-of-the-art implementation of Generic Join and Free Join methods, respectively, across acyclic and cyclic queries in standard query benchmarks.

Lay Summary

In this thesis, we introduce a unified architecture that integrates binary join and worst-case optimal join (WCOJ) algorithms, as well as both hash-based and sort-based WCOJ paradigms. Our proposed system consistently outperforms or matches state-of-the-art solutions across all hash-based, sort-based, and hybrid approaches. This flexibility allows for selecting the most efficient method tailored to the input data and the executing query.

Acknowledgements

I would like to thank my supervisor, Dr. Amir Shaikhha, for his guidance, patience, and support throughout this research journey, and I am grateful for the opportunity to learn under his supervision.

I owe my deepest thanks to my partner, whose support, encouragement, and understanding have been my constant source of strength. I am forever grateful for her presence by my side.

I would also like to acknowledge the financial support provided by RelationalAI, without which this research would not have been possible.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Amirali Kaboli

Contents

Figures and Tables

Figures

Tables

Chapter 1

Introduction

Efficient query processing is essential for the performance of modern database management systems, particularly as the complexity and size of data continue to grow. Join operations, which combine records from multiple tables, play a pivotal role in this process; however, traditional join algorithms often face significant efficiency challenges when processing complex queries that produce intermediate results much larger than the final output. The emergence of worst-case optimal join (WCOJ) algorithms Ngo (2018); Ngo, Porat, Ré, and Rudra (2012); Ngo, Ré, and Rudra (2014); Veldhuizen (2013) represents a significant advancement, offering asymptotically better performance by avoiding the enumeration of potentially exploding intermediate results. It can be the case for both cyclic and acyclic queries as opposed to the common belief that WCOJ is designed for cyclic queries.

There have been previous efforts Aberger et al. (2017); Freitag, Bandle, Schmidt, Kemper, and Neumann (2020); Mhedhbi and Salihoglu (2019) to adopt an approach where WCOJ algorithms are used for certain parts of a query while traditional join algorithms are applied to the rest. However, the use of two distinct algorithms within the same system introduces additional complexity, which has hindered the broader adoption of WCOJ methods. The current state-of-the-art system, and our main competitor in this work, is Free Join Wang, Willsey, and Suciu (2023), which aims to address this challenge by unifying WCOJ with traditional join methods. Free Join provides a platform capable of executing a wide range of join queries, offering performance benefits in both WCOJ and traditional join scenarios. However, despite these advances, Free Join supports only a specific class of WCOJ algorithms–hash-based approaches–limiting its coverage and flexibility in handling other algorithmic paradigms such as sort-based joins.

Our approach stands out by leveraging advancements in programming languages, specifically through the use of SDQL Shaikhha, Huot, Smith, and Olteanu (2022), an intermediate language designed for functional collection programming using semi-ring dictionaries. By employing SDQL as our intermediate representation, we can translate worst-case optimal join (WCOJ) queries into an intermediate representation that can be directly compiled into highly efficient C++ code. This gives our system a key advantage in flexibility and performance optimization.

The use of SDQL enables us to introduce a unified architecture that efficiently supports both traditional binary joins and WCOJ algorithms. Moreover, our system can handle hash-based and sort-based paradigms of WCOJ processing, significantly improving over state-of-the-art systems such as Free Join. Existing systems typically only focus on one approach – either hash-based or sort-based – while we provide support for both, ensuring that our system can adapt to various input data types and query execution scenarios. This broad capability enhances the versatility and overall performance of our system across a wide range of join queries. We demonstrate that our system not only matches but also outperforms the performance of the Free Join framework as the state-of-the-art.

Contributions. In this thesis, we present the following contributions:

1. A unified architecture that integrates both efficient binary join and WCOJ processing (Chapter 3).
2. A comprehensive set of optimizations that refine our initial implementation into a highly optimized system (Chapters 4.1 and 4.2).
3. A novel hybrid approach, along with support for sort-based WCOJ algorithms, leverages the strengths of both hash-based and sort-based paradigms (Chapter 4.3).
4. A detailed experimental evaluation of our system and the applied optimizations (Chapter 5). Specifically, we show that our method achieves speedups of up to $3.1\times$ and $4.8\times$ compared to the Generic Join and Free Join implementations within the Free Join framework, respectively.

Chapter 2

Background

A full conjunctive query is expressed in the form shown in Eq (2.1). In this notation, each term $R_i(x_i)$ is referred to as an atom, where R_i represents a relation name, and x_i is a tuple of variables. The query is considered full because the head variables x encompass all variables that appear in the atoms. We assume that selections have been pushed down to the base tables, meaning that each atom R_i may already incorporate a selection over a base relation. Similarly, projections and aggregations are performed only after the full join operation is executed, and hence, they are not explicitly included in Eq (2.1).

$$Q(x) : -R_1(x_1), \ldots, R_m(x_m) \tag{2.1}$$

Example. Throughout this thesis, we will make use of a conjunctive query referred to as $Q_{\clubsuit}$, denoted in Eq (2.2).

$$Q_{\clubsuit}(x,a,b) : -R(x,a), S(x,b), T(x) \tag{2.2}$$

The SQL query of $Q_{\clubsuit}$ is as below and its corresponding implementation using binary joins in SDQL and C++ is shown in Figure 2.1.

```
SELECT * FROM R, S, T
WHERE R.x = S.x AND R.x = T.x AND S.x = T.x
```

2.1 Generic Join

The Generic Join algorithm, first introduced in Ngo et al. (2014), represents the simplest worst-case optimal join algorithm. It builds upon the earlier Leapfrog Triejoin algorithm Veldhuizen (2013). The Generic Join algorithm computes the query Q from Eq (2.1) by executing a series of nested loops, where each loop iterates over a specific query variable. In particular, Generic Join arbitrarily selects a variable x, computes the intersection of all x-columns across the relations that contain x, and for each value θ in this intersection, it evaluates the residual

```
let S_ht =
  sum(<i, _> <- range(S.size))
    {S.x(i) -> {i -> 1}} in

let RS =
  sum(<R_i, _> <- range(R.size))
    let x = R.x(R_i) in
    if (x ∈ S_ht)
      let Sx = S_ht(x) in
      sum (S_i, _> <- Sx)
        {<x=x, a=R.a(R_i),
  b=S.b(S_i)> -> 1} in


let T_ht =
  sum(<i, _> <- range(T.size))
    {T.x(i) -> {i -> 1}} in


sum(<RS_i, _> <- range(RS.size))
  let x = RS.x(RS_i) in
  if (x ∈ T_ht)
    let Tx = T_ht(x) in
    sum (T_i, _> <- Tx)
      {<x=x, a=RS.a(RS_i),
  b=RS.b(RS_i)> -> 1}
```

(a) SDQL.

```
HT<int, HT<int, bool>> S_ht;
for (int i = 0; i < S.size; ++i)
  S_ht[S.x[i]][i] += 1;

HT<tuple<int, int, int>, int> RS;
for (int R_i = 0; R_i < R.size; ++R_i){
  auto x = R.x[R_i];
  if (S_ht.contains(x)) {
    auto &Sx = S_ht.at(x);
    for (auto &[S_i, S_v] : Sx)
      res[{x, R.a[R_i], S.b[S_i]}] +=
  1;
}}

HT<int, HT<int, bool>> T_ht;
for (int i = 0; i < T.size; ++i)
  T_ht[T.x[i]][i] += 1;

HT<tuple<int, int, int>, int> RST;
for (auto &[RS_i, RS_v] : RS) {
  auto x = get<0>(RS[RS_i]);
  if (T_ht.contains(x)) {
    auto &Tx = S_ht.at(x);
    for (auto &[T_i, T_v] : Tx)
      RST[{x, get<1>(RS[RS_i]),
  get<2>(RS[RS_i])}] += 1;
}}
```

(b) C++.

Figure 2.1: Implementation of $Q_{\clubsuit}$ based on traditional binary joins in SDQL and C++.

query $Q[x/\theta]$. In the residual query, every relation R containing x is replaced by $\sigma_{x=\theta}(R)$ (or equivalently $R[\theta]$). If the query Q contains k variables, the algorithm proceeds with k nested loops. In the innermost loop, Generic Join outputs a tuple consisting of constants derived from each iteration.

The Generic Join algorithm is provably worst-case optimal, achieving a time complexity of $O(n^{1.5})$, where n represents the maximum possible size of the output Atserias, Grohe, and Marx (2013); Ngo et al. (2014). In contrast, binary join algorithms can exhibit a time complexity of $O(n^2)$. While binary joins rely on hash tables for efficiency, Generic Join leverages a hash trie structure for each relation in the query. A hash trie is a tree structure where each node is either a leaf node or a hash map that associates each atomic attribute value with another node.

2.2 Free Join

A Free Join plan defines how the Free Join algorithm is executed, serving as a generalization and unification of both binary join plans and Generic Join plans Wang et al. (2023). In a left-deep linear plan for binary joins, the execution order is represented as a sequence of relations, where join attributes are implicitly determined by the shared attributes between relations. In contrast, a Generic Join plan outlines a sequence of variables and does not explicitly reference the relations, as joins are performed on any relation sharing a particular variable. A Free Join plan, however, allows for the joining of any number of variables and relations at each step, requiring both to be explicitly specified.

Formally, a Free Join plan is represented as a list $[\phi_1, \ldots, \phi_m]$, where each ϕ_k is a list of subatoms from the query Q, referred to as a node. The nodes must partition the query in the sense that for every atom $R_i(x_i)$ in the query, the set of all subatoms in all nodes must constitute a partitioning of $R_i(x_i)$. A subatom of an atom $R_i(x_i)$ is of the form $R_i(y)$, where y is a subset of the variables x_i. A partitioning of the atom $R_i(x_i)$ consists of subatoms $R_i(y_1), R_i(y_2), \ldots$, where the sets $y_1, y_2, \ldots$ form a partition of x_i.

To construct a Free Join plan, the system begins with an optimized binary join plan produced by a traditional cost-based optimizer, such as DuckDB Raasveldt (2022); Raasveldt and Mühleisen (2019); Raasveldt and Mühleisen (2020). It first decomposes a bushy plan into a set of left-deep plans. Each left-deep plan is then converted into an equivalent Free Join plan. After conversion, further optimization yields a plan that can range from a left-deep plan to a Generic Join plan.

Consider a plan derived from a straightforward conversion of the binary join plan for the clover query $Q_\clubsuit$ into a Free Join plan, as shown in Eq (2.3). To execute the first node, we iterate over each tuple (x, a) in R and use x to probe into S. For each successful probe, we proceed to the second node, iterating over each value b in $S[x]$, and then using x to probe into T.

$$[[R(x,a), S(x)], [S(b), T(x)]] \tag{2.3}$$

The plan corresponding to the Generic Join plan for the clover query $Q_\clubsuit$ is depicted in Eq (2.4). In this plan, execution starts by intersecting the sets $R.x \cap S.x \cap T.x$. For each x in the intersection, the values of a and b are retrieved from R and S, respectively, and their Cartesian product is computed. Additionally, after optimizing the naive plan from Eq (2.3), the resulting optimized Free Join plan for the clover query $Q_\clubsuit$ is shown in Eq (2.5).

$$[[R(x), S(x), T(x)], [R(a)], [S(b)]] \tag{2.4}$$

$$[[R(x,a), S(x), T(x)], [S(b)]] \tag{2.5}$$

2.3 SDQL

We utilize SDQL, a statically typed language capable of expressing relational algebra with aggregations, and functional collections over data structures such as relations using semi-ring dictionaries. SDQL can serve as an intermediate language for data analytics, enabling the translation of programs written in relational algebra-based languages into SDQL.

Figure 2.1 illustrates the corresponding SDQL program to execute $Q_{\clubsuit}$ using traditional binary joins, alongside its equivalent generated C++ code. In this example, we first join relations R and S (lines 1-11), followed by a join with relation T (lines 14-24). In SDQL, we use the `let` keyword to declare variables, such as a dictionary in line 1. The `sum` keyword enables iteration over dictionary entries, as demonstrated in line 2. Conditional logic is expressed with `if`, and membership is checked using the $\in$ operator to verify if an element exists in a dictionary (line 8). If an element exists, its associated value can be accessed using the $(\ldots)$ syntax, as shown in line 9. We employ `std::tuple` to implement records (line 11).

In addition to these basic and predefined syntaxes, we extended SDQL to meet our requirements by incorporating support for various dictionary types and underlying data structures for dictionary representation. The subsequent sections will explore these extensions in detail.

Chapter 3

System

The methodology, as depicted in Figure 3.1, employs a multi-stage pipeline architecture. The initial phase involves the transformation of a binary plan into a Free Join plan (Chapter 3.1). Subsequently, we proceed to generate a naive SDQL program corresponding to the Free Join plan (Chapter 3.2). Several optimizations are applied to enhance the performance of the initial SDQL program (Chapter 4). The final stage of the pipeline entails the generation of efficient C++ code derived from the SDQL program, facilitating efficient query execution (Chapter 3.3).

3.1 Planning

This approach begins by taking an optimized binary join plan as input and transforming it into a Free Join plan Wang et al. (2023). This transformation process is based on the methodology described in Chapter 2.2 and the Free Join paper, which we utilize for consistency and fair comparison. The system initially starts with an optimized binary plan and then converts it to an equivalent Free Join plan. The corresponding plans for the Generic Join and Free Join algorithms for $Q_{\clubsuit}$ would resemble those in Eq (2.4) and Eq (2.5), respectively.

3.2 SDQL Program Generation

At this stage, we take the Free Join plan from the previous step and generate an efficient SDQL program as an intermediate representation. This process is divided into two phases: trie creation and query execution.

Trie Creation Phase. We construct the tries that will be utilized during the query execution phase. Within SDQL, these tries function as nested hash maps, where each leaf node is a set of offsets into the base relation represented as a hash map with these offsets as the key and `true` as the value. Each internal level is a hash map that maps attribute values to the next level's hash map.

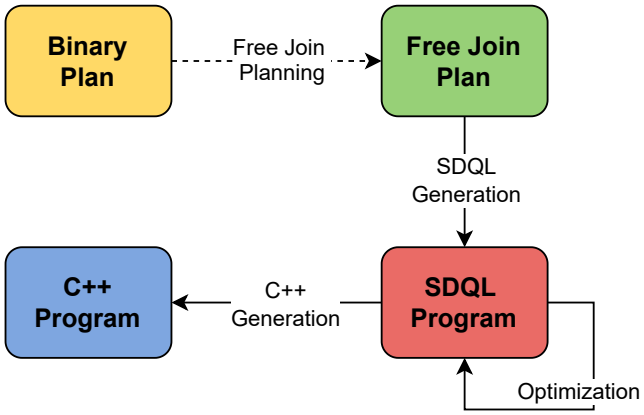


Figure 3.1: An overview of our system architecture.

For instance, as illustrated in lines 2-10 of Figure 3.2a and Eq (2.4), we need a trie for each of the relations R, S, and T in $Q_{\clubsuit}$, which align with the Generic Join algorithm. These tries are created from attribute x to the offsets of each relation, enabling access to other attributes in subsequent steps. In contrast, the Free Join implementation only requires building tries over relations S and T (lines 2-7 of Figure 3.3a) since we iterate directly over relation R.

Query Execution Phase. We generate an SDQL program that corresponds to the converted Free Join plan, as described in Chapter 3.1, utilizing the previously constructed tries. The execution of each node in the Free Join plan involves iterating over the first relation or its trie, depending on the plan, and using the attribute values to probe into each of the other tries.

In the example, for the first node in the plan for Generic Join $[R(x), S(x), T(x)]$, we iterate over R and use x values to probe into S and T. We do a similar process for the first node of the Free Join's plan $[R(x,a), S(x), T(x)]$. For each x value successfully probed in both, we proceed to execute the second node. This process is reflected in lines 14-17 and 11-15, corresponding to the first node of the plan in Figures 3.2a and 3.3a, respectively. The subsequent nodes represent the Cartesian product among the other attribute values to make the final results for a given x value. The translation of the nodes $[[R(a)], [S(b)]]$ are shown in lines 18-20 in Figure 3.2a, and the translation of $[S(b)]$ is shown in lines 16-17 in Figure 3.3a.

3.3 C++ Code Generation

The final component of our pipeline involves generating C++ code for SDQL programs, which is relatively straightforward, as illustrated in Figures 3.2 and 3.3. To enhance performance, summations that yield dictionaries are translated into loops that perform in-place updates. In addition to the transformations outlined in Chapter 2.3, the subsequent sections will demonstrate the translation of the extended data structures.

```
// Trie Creation
let R_trie =
  sum(<i, _> <- range(R.size))
    {R.x(i) -> {i -> 1}} in
let S_trie =
  sum(<i, _> <- range(S.size))
    {S.x(i) -> {i -> 1}} in
let T_trie =
  sum(<i, _> <- range(T.size))
    {T.x(i) -> {i -> 1}} in

// Query Execution

sum(<x, Rx> in R_trie)
  if (x ∈ S_trie && x ∈
 T_trie) then
    let Sx = S_trie(x) in
    let Tx = T_trie(x) in
    sum(<R_i, _> <- Rx)
      sum(<S_i, _> <- Sx)
        {<c0=x, c1=R.a(R_i),
 c2=S.b(S_i)> -> 1}
```

(a) SDQL.

```
// Trie Creation
HT<int, HT<int, bool>> R_trie;
for (int i = 0; i < R.size; ++i)
  R_trie[R.x[i]][i] += 1;
HT<int, HT<int, bool>> S_trie;
for (int i = 0; i < S.size; ++i)
  S_trie[S.x[i]][i] += 1;
HT<int, HT<int, bool>> T_trie;
for (int i = 0; i < T.size; ++i)
  T_trie[T.x[i]][i] += 1;

// Query Execution
HT<tuple<int, int, int>, int> res;
for (auto &[x, Rx] : R_trie)
  if (S_trie.contains(x) &&
 T_trie.contains(x)) {
    auto &Sx = S_trie.at(x);
    auto &Tx = T_trie.at(x);
    for (auto &[R_i, R_v] : Rx)
      for (auto &[S_i, S_v] : Sx)
        res[{x, R.a[R_i], S.b[S_i]}]
 += 1;
  }
```

(b) C++.

Figure 3.2: Generic Join implementation of $Q_{\clubsuit}$ in SDQL and C++.

```
// Trie Creation
let S_trie =
  sum(<i, _> <- range(S.size))
    {S.x(i) -> {i -> 1}} in
let T_trie =
  sum(<i, _> <- range(T.size))
    {T.x(i) -> {i -> 1}} in

// Query Execution

sum(<R_i, _> in range(R.size))
  let x = R.x(R_i) in
  if (x ∈ S_trie && x ∈
 T_trie) then
    let Sx = S_trie(x) in
    let Tx = T_trie(x) in
    sum(<S_i, _> <- Sx)
      {<c0=x, c1=R.a(R_i),
 c2=S.b(S_i)> -> 1}
```

(a) SDQL.

```
// Trie Creation
HT<int, HT<int, bool>> S_trie;
for (int i = 0; i < S.size; ++i)
  S_trie[S.x[i]][i] += 1;
HT<int, HT<int, bool>> T_trie;
for (int i = 0; i < T.size; ++i)
  T_trie[T.x[i]][i] += 1;

// Query Execution
HT<tuple<int, int, int>, int> res;
for (int R_i = 0; R_i < R.size;
 ++R_i) {
  auto x = R.x[R_i];
  if (S_trie.contains(x) &&
 T_trie.contains(x)) {
    auto &Sx = S_trie.at(x);
    auto &Tx = T_trie.at(x);
    for (auto &[S_i, S_v] : Sx)
      res[{x, R.a[R_i], S.b[S_i]}] +=
 1;
}}
```

(b) C++.

Figure 3.3: Free Join implementation of $Q_{\clubsuit}$ in SDQL and C++.

Chapter 4

Efficiency

In this section, we sequentially apply a series of optimizations to the naive implementation of $Q_{\clubsuit}$, building upon each previous one for better comprehension. Chapter 4.1 introduces dictionary specialization, optimizing the underlying data structures for dictionary representation. Chapter 4.2 discusses the early projection techniques employed to enhance performance. Finally, Chapter 4.3 presents our novel hybrid approach, alongside the support of sort-based WCOJ algorithms, to leverage the strengths of both hash-based and sort-based paradigms.

4.1 Dictionary Specialization

Dictionary specialization is a technique aimed at optimizing the data structures used to represent leaf nodes in tries, significantly improving both trie creation and query execution performance. When a dictionary is created in SDQL, it serves two primary purposes: lookup and iteration. When lookups are required, an underlying dictionary data structure is necessary. However, if the operation only involves iterating over the dictionary, we can store only the keys in a more efficient data structure. This subsection focuses on two specific optimizations we employed to enhance dictionary specialization in our system: `std::vector` and SmallVector.

4.1.1 Vector (O1)

Replacing hash maps with vectors for leaf nodes in tries can significantly improve performance by reducing the overhead of key-value pair operations. As outlined in Chapter 3.2, each leaf node in the tries was initially represented as a hash map that mapped offsets in the base relation to a constant boolean value (`true`). However, since only the keys of these hash maps are required, as demonstrated in Figures 3.2a and 3.3a, this structure can be optimized by converting the key-value pair representation into a list implementation that stores only the keys.

```
class VecDict {
  vector<T> vec; // SmallVector in SmallVecDict
  class Proxy {
    VecDict &vecdict;
    T key;
    void operator+=(int) {
      vecdict.vec.push_back(key);
  }};
  Proxy operator[](T key) {
    return Proxy(*this, key);
}};
```

Figure 4.1: `VecDict` **data structure.**

This optimization is applied in SDQL programs through the use of `@vec` annotation to specify the dictionary representation, as shown in Figure 4.2a. When this annotation is applied to a hash map, we employ the `VecDict` data structure, as shown in Figure 4.1, in C++, replacing the hash table that maps offsets to `true`, as illustrated in Figure 4.2b. This data structure acts as a wrapper around `std::vector`, providing a dictionary-like interface. Using a vector under the hood enhances performance by reducing the cost of both insertion and iteration, as operations in `std::vector` are generally less expensive than in hash tables.

4.1.2 SmallVector (O2)

SmallVector is a specialized data structure designed to function as a vector-like container optimized for small sequences. It improves performance by allocating storage on the stack, thus reducing the overhead of heap allocations and enhancing cache locality. This optimization is particularly advantageous when the vector contains only a small number of elements. SmallVector allocates a fixed number of elements on the stack, and when the number of elements exceeds this predefined size, it switches to heap allocation. This strategy offers a balance between performance and flexibility, and implementations of SmallVector are used in systems such as Rust Rust Crate Developers (n.d.) and LLVM LLVM Project (n.d.).

We implemented a custom version of SmallVector, as shown in Figure 4.3 in C++ to replace the underlying data structure of the leaf nodes used to store offsets. As previously discussed, when performing lookups on relation R for a given θ value, $R[\theta]$ typically contains a small number of elements. In such cases, SmallVector enhances performance by avoiding heap allocation for nodes with few elements during trie creation. Another application of this data structure occurs when building a trie on a unique attribute, such as primary keys, which is common in join operations. Since this attribute is unique, each value appears only once, resulting in a single offset per value. While a single variable could be used in this scenario, SmallVector effectively handles it.

```
// Trie Creation
let S_trie =
  sum(<i, _> <- range(S.size))
-   S.x(i) -> {i -> 1}} in
+   {S.x(i) -> @vec {i -> 1}} in
let T_trie =
  sum(<i, _> <- range(T.size))
-   {T.x(i) -> {i -> 1}} in
+   {T.x(i) -> @vec {i -> 1}} in

// Query Execution

sum(<R_i, _> in range(R.size))
  let x = R.x(R_i) in
  if (x ∈ S_trie && x ∈ T_trie)
 then
    let Sx = S_trie(x) in
    let Tx = T_trie(x) in
    sum(<S_i, _> <- Sx)
      {<c0=x, c1=R.a(R_i),
 c2=S.b(S_i)> -> 1}
```

(a) SDQL.

```
// Trie Creation
- HT<int, HT<int, bool>> S_trie;
+ HT<int, VecDict<int>> S_trie;
for (int i = 0; i < S.size; ++i)
  S_trie[S.x[i]][i] += 1;
- HT<int, HT<int, bool>> T_trie;
+ HT<int, VecDict<int>> T_trie;
for (int i = 0; i < T.size; ++i)
  T_trie[T.x[i]][i] += 1;

// Query Execution
HT<tuple<int, int, int>, int> res;
for (int R_i = 0; R_i < R.size;
 ++R_i) {
  auto x = R.x[R_i];
  if (S_trie.contains(x) &&
 T_trie.contains(x)){
    auto &Sx = S_trie.at(x);
    auto &Tx = T_trie.at(x);
-     for (auto &[S_i, S_v] : Sx)
+     for (auto &S_i : Sx)
      res[{x, R.a[R_i],
 S.b[S_i]}] += 1;
}}
```

(b) C++.

Figure 4.2: Impact of using Vector data structure for inner dictionaries which store offsets into base relations.

The interface of SmallVector closely mirrors that of `std::vector`, allowing it to be seamlessly integrated in contexts where `std::vector` is typically used, as shown in Figure 4.4. SmallVector manages dynamic memory allocation internally, abstracting the complexity from the user while delivering improved efficiency.

4.2 Early Projection/Aggregation

Early projection and aggregation are techniques that reduce the amount of data processed and stored during query execution by identifying and eliminating unnecessary columns as early as possible. This approach can significantly improve query performance by reducing memory usage, enhancing cache utilization, and accelerating join operations. In this subsection, we discuss three specific optimizations that we employed in our system.

```
class SmallVector {
  array<T, N> stack;
  vector<T> *heap;
  size_t size{0};
  void push_back(const T &value) {
    if (size++ < N)
      stack[size] = value;
    else {
      if (size++ == N) {
        heap = new vector<T>(
          stack.begin(),
 stack.end());
      }
      heap->push_back(value);
}}};
```

Figure 4.3: `SmallVector` **data structure.**

```
// Trie Creation
let S_trie =
  sum(<i, _> <- range(S.size))
-   {S.x(i) -> @vec {i -> 1}} in
+   {S.x(i) -> @smallvec(4) {i ->
 1}} in
let T_trie =
  sum(<i, _> <- range(T.size))
-   {T.x(i) -> @vec {i -> 1}} in
+   {T.x(i) -> @smallvec(4) {i ->
 1}} in
```

(a) SDQL.

```
// Trie Creation
- HT<int, VecDict<int>> S_trie;
+ HT<int, SmallVecDict<int, 4>>
   S_trie;
for (int i = 0; i < S.size; ++i)
  S_trie[S.x[i]][i] += 1;
- HT<int, VecDict<int>> T_trie;
+ HT<int, SmallVecDict<int, 4>>
   T_trie;
for (int i = 0; i < T.size; ++i)
  T_trie[T.x[i]][i] += 1;
```

(b) C++.

Figure 4.4: Impact of using SmallVector data structure for inner dictionaries which store offsets.

4.2.1 Dead Code Elimination (O3)

Dead code elimination is a powerful optimization technique that can significantly improve query performance by reducing the amount of data processed and stored during join operations. This technique systematically removes unnecessary columns during the join process, whether these columns originate from base relations or intermediate results. By identifying and eliminating attributes that are not required for subsequent operations or the final query output, the volume of data processed and stored throughout the query execution pipeline is minimized. This reduction accelerates join operations by decreasing memory usage and enhancing cache utilization, leading to overall improvements in query efficiency.

```
// Trie Creation
let S_trie =
  sum(<i, _> <- range(S.size))
    {S.x(i) -> @smallvec(4) {i -> 1}} in
let T_trie =
  sum(<i, _> <- range(T.size))
-   {T.x(i) -> @smallvec(4) {i -> 1}} in
+   {T.x(i) -> 1} in
```

(a) SDQL.

```
// Trie Creation
HT<int, SmallVecDict<int, 4>> S_trie;
for (int i = 0; i < S.size; ++i)
  S_trie[S.x[i]][i] += 1;
- HT<int, SmallVecDict<int, 4>> T_trie;
+ HT<int, int> T_trie;
for (int i = 0; i < T.size; ++i)
-   T_trie[T.x[i]][i] += 1;
+   T_trie[T.x[i]] += 1;
```

(b) C++.

Figure 4.5: Impact of redundant offsets elimination for join-only relations.

4.2.2 Eliminating Redundant Offsets (O4)

For relations that do not contribute to the final query output, it is unnecessary to store their offsets during trie construction, as we do not need to access other attributes from these relations. These relations are only utilized for joining and checking the existence of values for the attributes involved in the joins. Therefore, eliminating redundant offsets can reduce the overhead associated with trie construction.

In the clover query $Q_{\clubsuit}$, for example, relation T is used to join on attribute x with relations R and S, but it does not contribute any attributes to the final results. For such relations, the primary task is to verify the existence of x values from R, without accessing T's attributes. Therefore, storing offsets for a relation like T becomes unnecessary. To address this, we can optimize the hash map by replacing the value type from a vector-like data structure with an integer variable, as illustrated in Figure 4.5. This modification reduces the overhead associated with appending elements for relations involved solely in joins, thereby streamlining the trie creation process and enhancing overall efficiency.

4.2.3 Loop-Invariant Code Motion (O5)

By identifying and moving invariant expressions out of loops, loop-invariant code motion can significantly reduce the number of operations required for aggregation calculations.

In the context of calculating the left-hand side of Eq (4.1), we can observe that for each β_j, α_i is a constant value and can be moved outside the inner summation loop. Similarly, now for each α_i, the summation of all values in β is also a constant value, allowing it to be moved outside the outer summation loop. This optimization reduces the number of required multiplications from $n \times k$ to 1 and the number of summations from $n \times k$ to $n + k$, resulting in a significant

```
// Query Execution

sum(<R_i, _> in range(R.size))
  let x = R.x(R_i) in
  if (x ∈ S_trie && x ∈ T_trie)
  then
    let Sx = S_trie(x) in
    let Tx = T_trie(x) in
-   sum(<S_i, _> <- Sx)
-
  promote[min_sum](<c0=R.a(R_i),
  c1=S.b(S_i)>)
+   let R_mn = <c0=R.a(R_i)> in
+   let S_mn =
+     sum(<S_i, _> <- Sx)
+
  promote[min_sum](<c0=S.b(S_i)>)
  in
+   promote[min_sum](<c0=R_mn.c0,
  c1=S_mn.c0>)

```

(a) SDQL.

```
// Query Execution
- HT<tuple<int, int, int>, int> res;
+ tuple<int, int> res {INF, INF};
for (int R_i = 0; R_i < R.size;
 ++R_i) {
  auto x = R.x[R_i];
  if (S_trie.contains(x) &&
 T_trie.contains(x)) {
    auto &Sx = S_trie.at(x);
    auto &Tx = T_trie.at(x);
-   for (auto &S_i : Sx)
-     min_inplace(res, {R.a[R_i],
  S.b[S_i]});
+   tuple<int> R_mn {R.a[R_i]};
+   tuple<int> S_mn {INF};
+   for (auto &S_i : Sx)
+     min_inplace(S_mn,
  {S.b[S_i]});
+   min_inplace(res,
  {get<0>(R_mn), get<0>(S_mn)});
}}
```

(b) C++.

Figure 4.6: Impact of loop-invariant code motion on aggregation operations.

performance improvement.

$$\sum_{i=1}^{n}\sum_{j=1}^{k}(\alpha_i \times \beta_j) = \sum_{i=1}^{n}(\alpha_i \times (\sum_{j=1}^{k}\beta_j)) = (\sum_{i=1}^{n}\alpha_i) \times (\sum_{j=1}^{k}\beta_j) \tag{4.1}$$

As discussed in Chapter 2, projections and aggregations are not explicitly represented in Free Join plans. For instance, consider an aggregation on query $Q_{\clubsuit}$, which projects only the minimum values of attributes a and b. The naive implementation of these aggregations is shown in the deleted lines (with light red background) of Figure 4.6. In this approach, for each x value that satisfies the join conditions, $2 \times k$ `min` operations are performed, assuming the size of S's offsets is k. By applying loop-invariant code motion, we can move the minimum operation for attribute a outside the loop over S's offsets, reducing the number of minimum operations by k. This leaves only k operations to find the minimum value of b, plus two additional operations to update the final output. This optimization effectively reduces unnecessary operations and improves the overall efficiency of the aggregation process.

```
class SortedDict {
  // stores keys and values in std::vector
  // values of type VT are int or Range
  vector::iterator find(const KT &key) {
    /* std::lower_bound binary search */
}};
```

Figure 4.7: `SortedDict` **data structure.**

4.3 Sorting vs Hashing

The realm of worst-case optimal joins is characterized by two primary paradigms: hash-based approaches, exemplified by Umbra Freitag et al. (2020) and Free Join Wang et al. (2023), and sort-based approaches, such as Leapfrog Triejoin Veldhuizen (2013), EmptyHeaded Aberger et al. (2017), and LMFAO Schleich, Olteanu, Abo Khamis, Ngo, and Nguyen (2019). Our system is designed to efficiently support both paradigms, allowing for the execution of sort-based WCOJ algorithms alongside hash-based methods. For the sort-based approach, we assume that input relations are always provided in sorted order.

Figure 4.9 illustrates the implementation of the sort-based approach for query $Q_{\clubsuit}$ in both SDQL and C++. In our system, the `@st` annotation is used to specify that the dictionary is a sorted dictionary. For the C++ implementation, we developed a custom sorted dictionary data structure, as shown in Figure 4.7. Since we assume that the input relations are sorted by the attributes involved in the joins, insertions always occur at the end, or we update the last elements in the sorted dictionary. For lookups, we employ binary search to efficiently locate a given key among the sorted keys and return its corresponding value.

Another optimization we employed in this case is the use of the `@range` annotation. When a relation is sorted by an attribute x, all occurrences of each x value appear consecutively. Instead of storing all occurrence offsets for each x value in a vector-like structure, we optimize by only keeping the first and last offsets of this consecutive block of elements, as shown in in Figure 4.8. To insert an offset into this structure, we simply update the right bound of the range. During iteration, we can efficiently loop from the left bound to the right bound, reducing both storage overhead and iteration complexity.

As discussed earlier, we decompose a bushy plan into a set of left-deep plans, and each left-deep plan produces an intermediate result. There is no guarantee that these intermediate results will remain sorted. For such cases, we must first sort the intermediate results before using them in trie creation with our sorted dictionary data structure. To reduce the overhead of sorting, we employ a hybrid approach by using a hash table for each intermediate result, bypassing the need for sorting them. While binary search in a sorted dictionary is less efficient

```
class Range {
  size_t left;
  size_t right;
  class Proxy {
    Range &range;
    void operator+=(int) { ++range.right; }
  };
  Proxy operator[](size_t const idx) {
    if ( /* is first access */ )
      left = right = idx;
    return Proxy(*this);
}};
```

Figure 4.8: `Range` **data structure.**

```
// Trie Creation
let S_trie =
  sum(<i, _> <- range(S.size))
    @st {S.x(i) -> @range {i ->
 1}} in
let T_trie =
  sum(<i, _> <- range(T.size))
    @st {T.x(i) -> @range {i ->
 1}} in
```

(a) SDQL.

```
// Trie Creation
SortedDict<int, Range> S_trie;
for (int i = 0; i < S.size; ++i)
  S_trie[S.x[i]][i] += 1;
SortedDict<int, Range> T_trie;
for (int i = 0; i < T.size; ++i)
  T_trie[T.x[i]][i] += 1;
```

(b) C++.

Figure 4.9: Sort-based implementation of the trie creation phase of $Q_{\clubsuit}$ in SDQL and C++.

than lookups in hash tables, the sorted dictionary proves advantageous during the trie creation phase, which is often the time-consuming part of a query. This allows the trie creation phase for intermediate results to remain more efficient, even when utilizing sorted dictionaries for base relations.

Chapter 5

Experiments

We implemented our system in a three-step pipeline. First, we take a binary join plan, produced and optimized by DuckDB, converting it into a Free Join plan Wang et al. (2023). Then, we translate the Free Join plan into an SDQL program, which serves as our intermediate representation, and apply various optimizations. Finally, we generate C++ code from the optimized SDQL program to execute the query.

We compare our approach against the Free Join framework Wang et al. (2023) on both Generic Join and Free Join implementations, recognizing it as the state-of-the-art system that outperforms in-memory databases such as DuckDB Raasveldt (2022); Raasveldt and Mühleisen (2019); Raasveldt and Mühleisen (2020). For an apple-to-apple comparison, we use the same query plans as the Free Join framework Wang et al. (2023). To evaluate performance, we use the widely adopted Join Order Benchmark (JOB) Leis et al. (2015) and the LSQB benchmark Mhedhbi, Lissandrini, Kuiper, Waudby, and Szárnyas (2021). Three research questions guide our evaluation:

1. How does our system compare to the Generic Join and Free Join implementations of Free Join framework? (Chapter 5.2)
2. What is the impact of optimizations we employed? (Chapter 5.3)
3. How do the hash-based and sort-based approaches perform in our system? (Chapter 5.4)

5.1 Setup

Both the JOB and LSQB benchmarks are primarily focused on evaluating join performance. The JOB benchmark consists of 113 acyclic queries, with an average of 8 joins per query, while the LSQB benchmark includes a mix of cyclic and acyclic queries. Each query in both benchmarks involves base-table filters, natural joins, and a simple group-by operation at the end. JOB operates on real-world data from the IMDB dataset, whereas LSQB uses synthetic data. For a fair comparison, we executed all benchmarks on the query set reported by the Free Join framework, which serves as our competitor. The only exception is query Q3 from the LSQB benchmark, which we excluded as the results are not reproducible using the Free Join framework's open-source implementation.

All experiments were conducted on a MacBook Pro running macOS 15.0.1, equipped with an Apple M1 Max chip and 64GB of LPDDR5 RAM. Each experiment was executed 5 times and the average run times were reported. All systems were configured to run in single-threaded mode and operate entirely in main memory. We employed an efficient hash table implementation in C++ known as phmap Popovitch (2024). All code was compiled using Clang 18.1.8 with the following flags:

```
-std=c++17 -O3 -march=native -mtune=native -Wno-narrowing -ftree-vectorize
```

5.2 Performance Comparison

Our first set of experiments compares the performance of our system for Generic Join and Free Join algorithms against the Free Join framework on both JOB and LSQB benchmarks.

5.2.1 JOB

Figures 5.1 and 5.2 presents a run time comparison of our system with the Free Join framework, evaluating both the Generic Join and Free Join algorithms on JOB queries. Since our system does not currently support vectorization, Figure 5.2a illustrates the performance of our system relative to the non-vectorized version of the Free Join framework, which employs the same underlying algorithm. We also used the hash-based approach of our system in Figure 5.1.

The majority of data points for Generic Join and both non-vectorized and vectorized versions of Free Join algorithms appear below the diagonal, indicating that our system outperforms the Free Join framework in these cases. This suggests that, despite the lack of vectorization support, our system outperforms the performance of the Free Join framework.

On average (geometric mean), our system demonstrates a speedup of $1.49\times$ and $1.42\times$ over the Free Join framework for the Generic Join and Free Join algorithms, respectively, and achieves a $2.70\times$ performance improvement over the non-vectorized version of Free Join. The maximum speedups observed are $3.14\times$ for Generic Join and $4.78\times$ for Free Join, while the minimum speedups are $0.71\times$ (40% slowdown) and $0.30\times$ ($3.33\times$ slowdown), respectively.

As discussed in Chapter 3.2, our system requires that tries be fully constructed before query execution begins, meaning the data structure we currently employ does not support lazy evaluation, unlike Free Join Wang et al. (2023). However, as outlined in Chapter 3.1, we utilize the same execution plans produced and used by the Free Join framework. In these plans, when a relation appears as the first relation in a node, we iterate over its offsets to access its attribute values. At this stage, all attribute values for that relation are made available to subsequent nodes in the execution plan, thereby eliminating the need for further iterations or lookups.

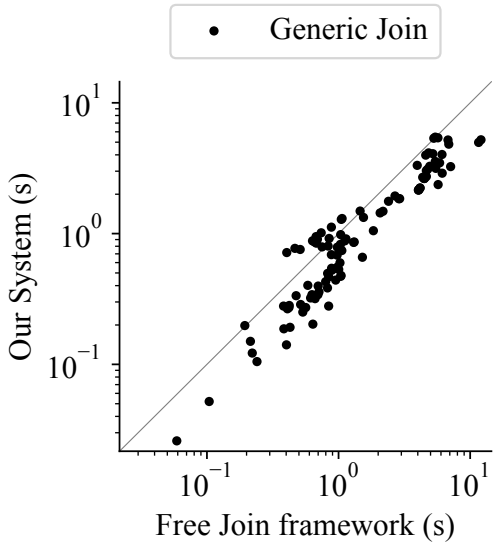
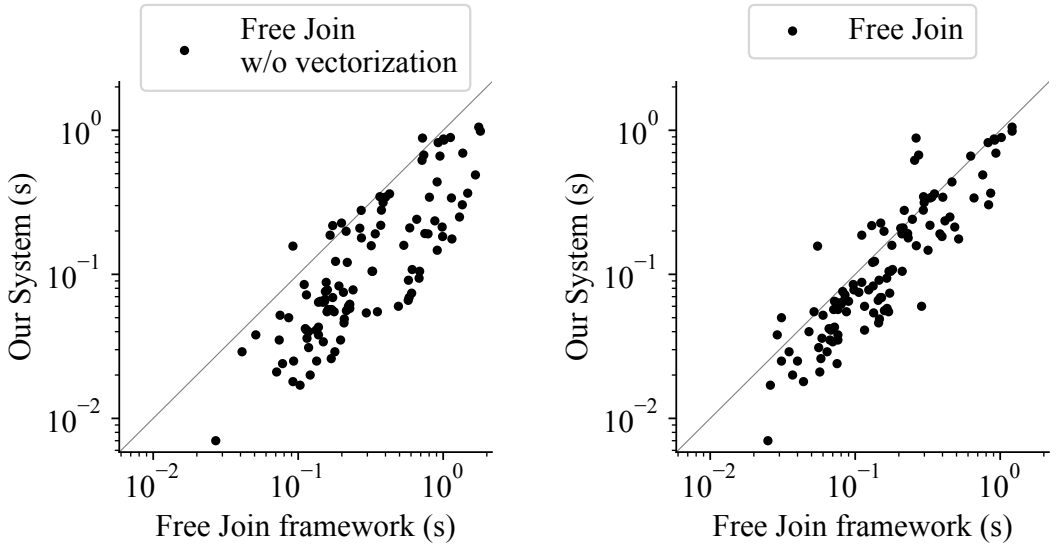


Figure 5.1: Run time comparison of Generic Join on JOB. Each point compares the run time of a query on our system and Free Join framework. Each point below the diagonal line represents a query for which our system is faster.

Once we identify the first node where each relation is used for iteration, we construct a trie with levels corresponding to the attributes of that relation that appeared in earlier nodes, since lookups for those attribute values are required. For all queries in the JOB benchmark, each relation involves at most one attribute lookup before iteration. This means that each relation is either used for iteration or accessed first for lookups over a single attribute, followed by iteration over the offsets linked to that attribute. Consequently, our approach behaves similarly to leveraging lazy data structures, as the construction of the first level of tries for non-iterated relations is necessary, and lookups over these relations are guaranteed.

5.2.2 LSQB

Figure 5.3 presents a performance comparison between our system and the Free Join framework for both Generic Join and Free Join algorithms on LSQB queries. Each line in the figure represents a query executed across scaling factors of 0.1, 0.3, 1, and 3. It is important to note that the Free Join framework encountered an error when running Q3, and we were unable to reproduce its results for this query.



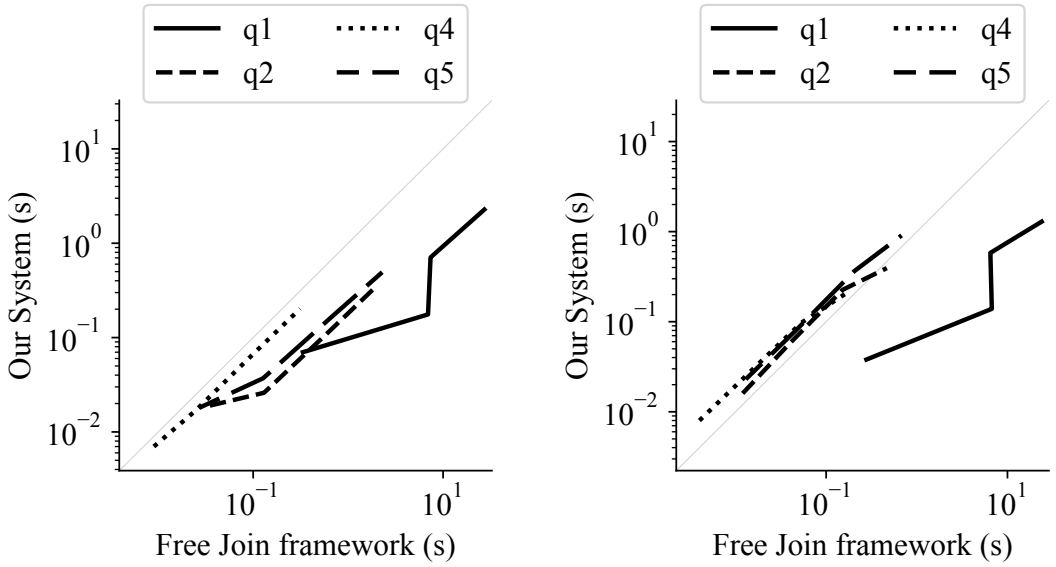
(a) w/o vectorization.

(b) w/ vectorization.

Figure 5.2: Run time comparison of Free Join on JOB. Each point compares the run time of a query on our system and Free Join framework. Figures 5.2a and 5.2b compare our system's Free Join implementation with the Free Join framework without and with vectorization. Each point below the diagonal line represents a query for which our system is faster.

For Q2, which is a cyclic query, our system outperforms the Free Join framework across scaling factors, achieving speedups of up to $5.67\times$ (on average $4.10\times$) for the Generic Join algorithm and a comparable run time for the Free Join algorithm. Contrary to our discussion about JOB queries in Chapter 5.2.1, there is a relation in Q2 that necessitates a trie with a depth greater than one. While our approach to trie construction in this scenario is less efficient than using lazy data structures, our system still demonstrates superior performance, with a significant performance gap compared to the Free Join framework.

For acyclic queries, while the performance of our Free Join implementation is comparable to the Free Join framework for Q4 and Q5, our system is up to $4.63\times$ (on average $3.13\times$) and $1.54\times$ (on average $1.44\times$) faster for the Generic Join algorithm, respectively. A significant performance improvement is observed for Q1, where our system achieves speedups of up to $39.41\times$ (on average $12.37\times$) for Generic Join and $48.37\times$ (on average $16.43\times$) for Free Join compared to the Free Join framework. We investigated the $48\times$ speedup, which occurs for scaling factor 0.3, and found it to show up repeatedly across benchmark runs. While we recognize the potential for a $10\times$ speedup through factorization in the Free Join framework, we were unable to reproduce their results. Even if we assume the Free Join framework achieves this speedup, our system would still maintain a considerable performance advantage based on the aforementioned speedups.



(a) Generic Join.

(b) Free Join.

Figure 5.3: Runtime comparison on LSQB. Each line is a query running on increasing scaling factors (0.1, 0.3, 1, 3) and compares our system and Free Join framework. Figure 5.3a compares the Generic Join implementation of each system. Figure 5.3b compares our system's Free Join implementation with Free Join framework.

Overall, this substantial gap is primarily attributed to the early projection and aggregation optimizations integrated into our system. Unlike the JOB queries, in LSQB, the output size before aggregation is significantly larger than the input size, resulting in a considerable amount of time spent on output construction. Our system mitigates this overhead by pushing projection and aggregation earlier in the query execution process.

5.3 Impact of Optimisations

As discussed in Chapter 4, we implemented a series of optimizations to enhance the efficiency of our naive implementation. Figure 5.4 illustrates the cumulative effect of these optimizations, showing the distribution of performance improvements relative to the Free Join framework for JOB queries. Initially, without any optimizations, our naive implementation of the Free Join algorithm was $2.11\times$ slower than the Free Join framework. Each subsequent optimization progressively narrowed this gap, contributing to the overall performance gains observed in our system.

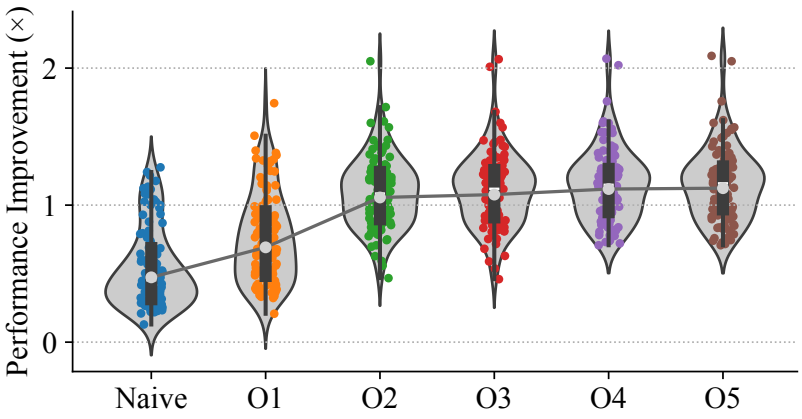


Figure 5.4: Impact of optimizations. Each point shows the performance improvement of a query over the Free Join framework. Each violin is the distribution of the performance improvements for all queries after applying the given optimization. The gray line shows the geometric mean for each optimization.

After applying O1, we observe an improvement in performance, though our system remains slower than the Free Join framework. With the application of O2, our system, despite lacking support for lazy data structures and vectorization, slightly outperforms the Free Join framework with a $1.056\times$ speedup. Up to this point, the impact of each optimization is clearly visible in Figure 5.4, and these optimizations are also available in the Free Join framework.

In the violin plot for O3, the lower part of the distribution (below the median) becomes thinner, while the upper part thickens, indicating a shift toward better performance. Additionally, we observe two data points with more than $2\times$ speedup rather than one in the previous optimization, and our overall speedup increases to $1.077\times$. With the application of O4, the tail of the O3 distribution is eliminated, resulting in an increased average speedup of $1.117\times$.

O5 further improves the performance of all queries slightly compared to O4. Ultimately, our fully optimized implementation achieves a $1.124\times$ speedup, which is $2.38\times$ faster than the naive implementation and 6.5% faster than O2. The subtle speedups in O3, O4, and O5 are attributed to the fact that trie construction dominates the overall run time for most of the queries. However, these optimizations are built on top of earlier optimizations targeting trie creation and only focus on improving the query execution phase.

We analyzed the impact of optimizations on a representative subset of queries, starting with our naive implementation and progressively applying each optimization. The results, shown in Figure 5.5, demonstrate that all optimizations introduced in Chapter 4 contribute positively to their respective scenarios. The O1 and O2 optimizations highlight the benefits of using `std::vector` and SmallVector, which enhance the performance of all selected queries. The

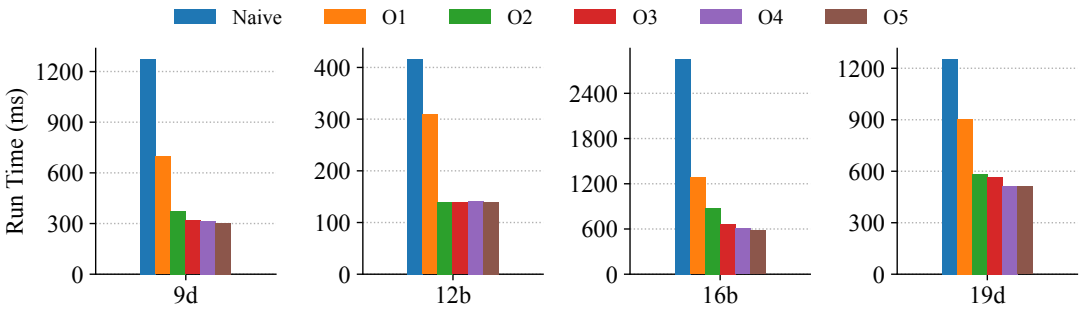


Figure 5.5: Ablation study. Each bar shows the run time of a query in our system after applying its corresponding optimization. O1: `std::vector`. O2: SmallVector. O3: Dead Code Elimination. O4: Eliminating Redundant Offsets. O5: Loop-Invariant Code Motion.

O3 optimization adds the Dead Code Elimination, affecting queries 9d and 16b. The O4 optimization further improves the previous ones by eliminating redundant offsets, which affects queries 16b and 19d. Finally, in O5, we apply Loop-Invariant Code Motion, which improves the performance of queries 9d and 16b.

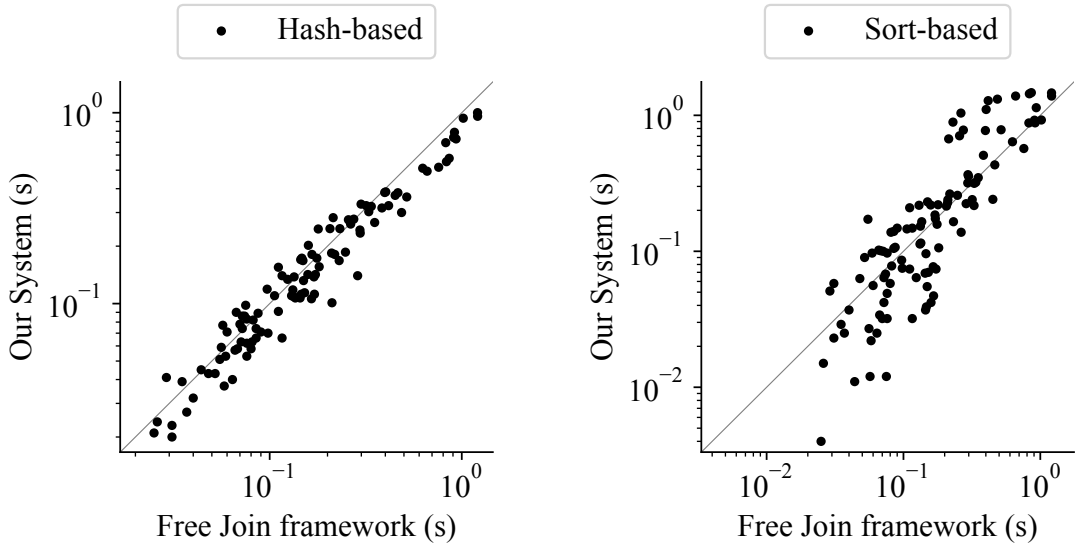
5.4 Hash-based vs Sort-based Performance

5.4.1 End-to-end benchmarks

In Figure 5.6a, illustrates the performance of the hash-based approach in our system. Data points, which represent the comparison between our hash-based implementation and Free Join, cluster around the diagonal. This suggests that the hash-based approach of our system matches and slightly outperforms the performance of the Free Join framework. Specifically, our system achieves a better performance up to $2.09\times$ (on average $1.12\times$) than the Free Join framework.

As discussed in Chapter 4.3, our system also supports the sort-based paradigm of worst-case optimal join (WCOJ) algorithms. For this class of algorithms, we assume that the input data is always provided in sorted order. Figure 5.6b presents the performance of our sort-based implementation for all JOB queries in comparison to the Free Join framework. Our sort-based approach demonstrates performance improvements of up to $6.25\times$ (on average $1.07\times$). As can be realized, these two approaches are algorithmically distinct, which explains why the data points are not concentrated around the diagonal in Figure 5.6b.

As mentioned earlier, even when input data is sorted, there is no guarantee that intermediate results will remain sorted. In such cases, we must sort these intermediate results before constructing their trie using a sorted dictionary, which can lead to significant overhead in the overall execution time. To address this, we introduce a novel hybrid approach that utilizes



(a) Hash-based.

(b) Sort-based.

Figure 5.6: Run time comparison between sort- and hash-based approaches and Free Join on JOB. Figure 5.6a compares the performance of the hash-based approach implemented in our system. Figure 5.6b compares the performance of the sort-based approach.

sorted dictionaries for base relations that are already sorted, while using hash tables for intermediate results. This eliminates the need to sort intermediate results. Using this hybrid approach, we achieve superior performance over the Free Join framework for almost all queries in the JOB benchmark, as shown in Figure 5.2b. Specifically, our hybrid approach demonstrates a performance improvement of up to $4.78\times$ (on average $1.42\times$) compared to Free Join.

For most queries in the JOB benchmark, trie creation is the most time-consuming aspect when using hash tables. By employing sorted dictionaries via `SortedDict` and `Range` data structures–where only the first and last offsets of an element's occurrences are stored instead of a vector-like structure–we can significantly improve run time performance. This approach reduces the overhead associated with hash tables, such as allocation, insertions, and updates. However, the use of binary search for lookups in sorted dictionaries can slow down query execution for cases where a significant portion of the run time is spent on the query execution phase itself, as seen in the queries above the diagonal in Figure 5.2b. The efficiency of each approach—hash-based or sort-based—depends on both the input data and the specific query being executed. However, our system provides the flexibility to utilize any of these approaches, enabling the efficient execution of any query on any dataset.

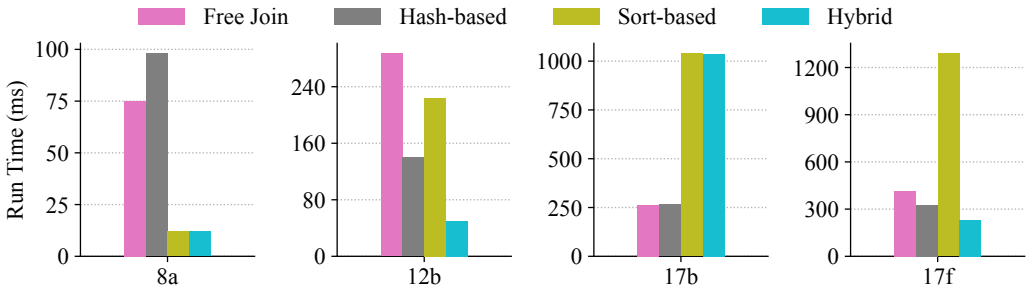


Figure 5.7: Run time comparison among Free Join and the hash-based, sort-based, and hybrid implementations in our system. Each bar shows the performance of an alternative on the given query.

Figure 5.7 highlights the run time of a representative subset of queries, providing deeper insights into the scenarios where each approach–hash-based, sort-based, or hybrid–performs better. For instance, query 12b, one of the points above the $2\times$ line in Figure 5.4, benefits from the hybrid approach, resulting in the highest speedups among all JOB queries. This improvement is largely due to the efficient handling of intermediate results using hash tables, which avoids the overhead of sorting. However, when dealing with smaller relations (after applying filters), sorting is relatively fast. In such cases, the hybrid approach's advantages may not fully offset the overhead introduced by hash tables. Query 8a is an example of this; as shown in Figure 5.7, the sort-based solution offers the best performance due to the small size of the relations, making sorting more efficient.

Queries 17b and 17f provide an interesting comparison, as they share the same joins but apply different filters to their base relations. Both queries slow down significantly with the sort-based approach, largely because they involve a large number of lookups where binary search negates the speedup gained from trie creation. However, the hybrid approach manages to compensate for the performance in query 17f but not in 17b. This disparity stems from the order in which lookups are applied. In query 17b, the plan probes a base relation first, followed by an intermediate result. Even with a hash table for the intermediate result, binary search is still used for all elements. In contrast, query 17f first probes the intermediate result using the hash table, where lookups are performed in a constant time. Only for the elements that find a match in the intermediate result does the query then perform binary searches on the base relation, reducing the number of $O(\log n)$ lookups, resulting in a more efficient execution.

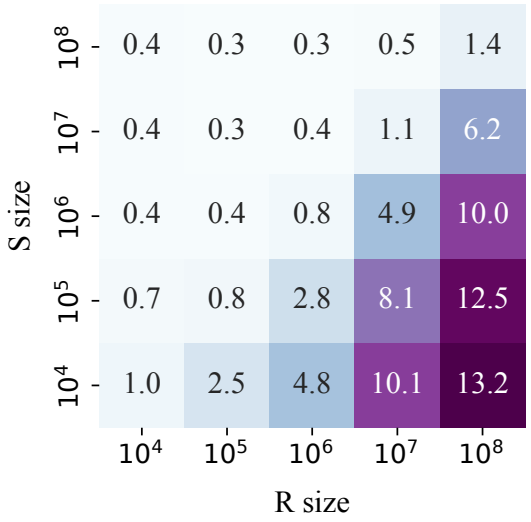


Figure 5.8: Run time comparison between hash-based and sort-based implementations of a join on relations R and S with varying sizes. Relation R is used for dictionary creation, while S is used for iteration and lookups. Each value reflects the relative performance of the sort-based approach compared to the hash-based implementation.

5.4.2 Microbenchmarks

Figure 5.8 presents a run time comparison between hash-based and sort-based implementations for joining two relations, R and S, across various sizes. In this setup, R is the relation for dictionary creation (build phase), while S is used for iteration and lookups (probe phase). The results highlight the conditions under which the sort-based (and hybrid) approaches provide advantages based on the size of the joining relations. When R and S are of similar magnitude, the performance difference between the approaches is minimal. However, the sort-based implementation may slow down by up to $2.5\times$ when S is substantially larger, but it achieves up to a $13.2\times$ speedup when S, the relation used for iteration and lookups, is smaller.

The optimal strategy to intersect $R_1.x \cap R_2.x \cap \ldots$ is to iterate over the smallest relation while probing each of the others. Free Join also adopts this approach in generating plans, which should theoretically enable sort-based approaches to perform comparably or even better. However, two factors hinder this outcome for some queries. First, when joining more than two relations, the number of lookups may exceed the size of the smallest relation since each iteration involves multiple lookups.

Query	Free Join Run Time	Ours - Default Plan Run Time	Ours - Default Plan Speedup	Ours - Revised Plan Run Time	Ours - Revised Plan Speedup
3b	92	182	$0.51\times$	47	$1.96\times$
17b	720	1037	$0.69\times$	124	$5.81\times$
15b	173	243	$0.71\times$	114	$1.52\times$
3a	199	255	$0.78\times$	137	$1.45\times$
14a	166	212	$0.78\times$	70	$2.37\times$
15a	272	307	$0.89\times$	192	$1.42\times$

Table 5.1: Run time comparison of our hybrid approach before and after plan revisions. The performance columns show the run time of our system using the default and revised plans. The speedup columns show the relative performance of our system over the non-vectorized version of Free Join.

Second, as shown in Table 5.1 (queries above the diagonal in Figure 5.2a), we identified cases where the smallest relation was not chosen for iteration. As demonstrated in Figure 5.8, this is not the scenario where the sort-based (or hybrid) approach significantly impacts run time. By revising these plans to consistently iterate over the smallest relation based on base relation cardinalities, we achieved speedups of up to $8.36\times$ over the default plans and $5.81\times$ over the corresponding Free Join implementation.

Chapter 6

Conclusion and Future Work

In this paper, we introduce a unified architecture that integrates binary join and worst-case optimal join (WCOJ) algorithms, as well as hash-based and sort-based WCOJ methods. Our proposed system consistently outperforms or matches state-of-the-art solutions across all hash-based, sort-based, and hybrid approaches. This flexibility allows for selecting the most efficient method tailored to the input data and the executing query.

For future research, we envision four primary directions. First, our system has limitations compared to the state-of-the-art, particularly Free Join Wang et al. (2023), as previously discussed. Enhancing performance through support for lazy data structures and vectorization is one area for further improvement. The second direction involves adding parallelism to our system. For instance, the query execution phase could be processed in chunks, as everything, except the final results, is read-only, requiring only the merging of final results. Similarly, parallelism could be exploited during the trie creation phase in the sort-based approach, since merging sorted dictionaries is straightforward.

We currently use binary search for lookups in sorted dictionaries, which has a time complexity of $O(\log n)$, asymptotically less efficient than the corresponding operation in hash tables. Another area of improvement is implementing hinted lookups Shaikhha, Ghorbani, and Shahrokhi (2023) for sorted dictionaries, which would allow us to achieve the same amortized time complexity as hash tables when the input data used for lookups is also already sorted.

Currently, our system utilizes the query optimizer developed by DuckDB and Free Join. However, this optimizer is not ideally suited for our system, as existing optimizers are typically designed to focus exclusively on either hash-based or sort-based approaches. An integrated optimizer tailored to our architecture could potentially generate more efficient query plans based on the specific algorithm employed. A notable example of this can be seen in the performance comparison between queries 17b and 17f. While the hash-based approach optimizes the order of lookups solely based on the relation sizes, the hybrid approach introduces an additional parameter that must be considered to produce an optimal execution plan.

Bibliography

Aberger, C. R., Lamb, A., Tu, S., Nötzli, A., Olukotun, K., & Ré, C. (2017, oct). Emptyheaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, *42*(4). Retrieved from https://doi.org/10.1145/3129246 doi: 10.1145/3129246

Atserias, A., Grohe, M., & Marx, D. (2013). Size bounds and query plans for relational joins. *SIAM Journal on Computing*, *42*(4), 1737-1767. Retrieved from https://doi.org/10.1137/110859440 doi: 10.1137/110859440

Freitag, M., Bandle, M., Schmidt, T., Kemper, A., & Neumann, T. (2020, jul). Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endow.*, *13*(12), 1891–1904. Retrieved from https://doi.org/10.14778/3407790.3407797 doi: 10.14778/3407790.3407797

Leis, V., Gubichev, A., Mirchev, A., Boncz, P., Kemper, A., & Neumann, T. (2015, November). How good are query optimizers, really? *Proc. VLDB Endow.*, *9*(3), 204–215. Retrieved from https://doi.org/10.14778/2850583.2850594 doi: 10.14778/2850583.2850594

LLVM Project. (n.d.). *Llvm smallvector documentation.* https://llvm.org/doxygen/classllvm_1_1SmallVector.html. (Accessed: 2024-10-15)

Mhedhbi, A., Lissandrini, M., Kuiper, L., Waudby, J., & Szárnyas, G. (2021). Lsqb: a large-scale subgraph query benchmark. In *Proceedings of the 4th acm sigmod joint international workshop on graph data management experiences & systems (grades) and network data analytics (nda).* New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3461837.3464516 doi: 10.1145/3461837.3464516

Mhedhbi, A., & Salihoglu, S. (2019, jul). Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.*, *12*(11), 1692–1704. Retrieved from https://doi.org/10.14778/3342263.3342643 doi: 10.14778/3342263.3342643

Ngo, H. Q. (2018). Worst-case optimal join algorithms: Techniques, results, and open problems. In *Proceedings of the 37th acm sigmod-sigact-sigai symposium on principles of database systems* (p. 111–124). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3196959.3196990 doi: 10.1145/3196959.3196990

Ngo, H. Q., Porat, E., Ré, C., & Rudra, A. (2012). Worst-case optimal join algorithms: [extended abstract]. In *Proceedings of the 31st acm sigmod-sigact-sigai symposium on principles of database systems* (p. 37–48). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2213556.2213565 doi: 10.1145/2213556.2213565

Ngo, H. Q., Ré, C., & Rudra, A. (2014, feb). Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.*, *42*(4), 5–16. Retrieved from https://doi.org/10.1145/2590989.2590991 doi: 10.1145/2590989.2590991

Popovitch, G. (2024, January). *The Parallel Hashmap C++ library.* Retrieved from https://github.com/greg7mdp/parallel-hashmap

Raasveldt, M. (2022). Duckdb - a modern modular and extensible database system. In *Cdms@vldb*. Retrieved from https://api.semanticscholar.org/CorpusID:252384081

Raasveldt, M., & Mühleisen, H. (2019). Duckdb: an embeddable analytical database. In *Proceedings of the 2019 international conference on management of data* (p. 1981–1984). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3299869.3320212 doi: 10.1145/3299869.3320212

Raasveldt, M., & Mühleisen, H. (2020). Data management for data science - towards embedded analytics. In *Conference on innovative data systems research.* Retrieved from https://api.semanticscholar.org/CorpusID:210712240

Rust Crate Developers. (n.d.). *Smallvec: A rust crate for small vector optimization.* https://docs.rs/smallvec/. (Accessed: 2024-10-15)

Schleich, M., Olteanu, D., Abo Khamis, M., Ngo, H. Q., & Nguyen, X. (2019). A layered aggregate engine for analytics workloads. In *Proceedings of the 2019 international conference on management of data* (p. 1642–1659). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3299869.3324961 doi: 10.1145/3299869.3324961

Shaikhha, A., Ghorbani, M., & Shahrokhi, H. (2023). Hinted Dictionaries: Efficient Functional Ordered Sets and Maps. In K. Ali & G. Salvaneschi (Eds.), *37th european conference on object-oriented programming (ecoop 2023)* (Vol. 263, pp. 28:1–28:30). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik. Retrieved from https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2023.28 doi: 10.4230/LIPIcs.ECOOP.2023.28

Shaikhha, A., Huot, M., Smith, J., & Olteanu, D. (2022). Functional collection programming with semi-ring dictionaries. *Proceedings of the ACM on Programming Languages*, *6*(OOPSLA1), 1–33.

Veldhuizen, T. L. (2013). *Leapfrog triejoin: a worst-case optimal join algorithm.* Retrieved from `https://arxiv.org/abs/1210.0481`

Wang, Y. R., Willsey, M., & Suciu, D. (2023, jun). Free join: Unifying worst-case optimal and traditional joins. *Proc. ACM Manag. Data*, *1*(2). Retrieved from `https://doi.org/10.1145/3589295` doi: 10.1145/3589295