A Modern DBMS for Research and Fast Prototyping

Immanuel Haffner    Jens Dittrich
January 9, 2023 @ CIDR
Saarland University
Saarland Informatics Campus
Database of Databases

Discover and learn about 877 database management systems

Most Recent

- FeatureBase
- EraDB
- Memgraph
- CeresDB
- CnosDB

Most Viewed

- levelDB
- BoltDB
- NeDB
- BTDB
- LMDB

Most Edited

- OCEANBASE
- CrateDB
- Solr
- CeresDB
- CnosDB
Database of Databases

877 database systems listed
Refine search to **academic** or **educational** projects with a **relational** data model.

Found 34 databases
Refine search to **academic** or **educational** projects with a **relational** data model.

Still contains some popular open-source projects, e.g. **POSTGRESQL**, **DuckDB**, and **NOISEPAGE**.
Add **Code Generation** to search criteria.

Search name, keywords, features... Search

Project Types: Academic  Project Types: Educational  Data Model: Relational
Query Compilation: Code Generation

Found 2 databases
Add **Code Generation** to search criteria.

Search name, keywords, features...

[Search]

*Project Types: Academic* [x]  *Project Types: Educational* [x]  *Data Model: Relational* [x]  *Query Compilation: Code Generation* [x]

Found 2 databases

Only **NOISEPAGE** (open source) and **UMBRA** (closed source) remain.
Status Quo of Open Source DBMSs?

Brief Summary

- PostgreSQL remains the top dog.
- A few emerging systems bring state-of-the-art technologies to open source, e.g., DuckDB and NoisePage.
- Still, much research remains proprietary / closed source, e.g., HyPer and Umbra.

Subjective Problem

- It is difficult for researchers and developers to get started with the available open source projects.
- Lack of documentation.
- Many built-in assumptions or design decisions.
Brief Summary

- POSTGRESQL remains the top dog.

Still, much research remains proprietary / closed source, e.g. HyPer and Umbra.
Status Quo of Open Source DBMSs?

**Brief Summary**

- POSTGRESQL remains the top dog.
- A few emerging systems bring state-of-the-art technologies to open source, e.g. DUCKDB and NOISEPAGE.
Brief Summary

- **POSTGRESQL** remains the top dog.
- A few emerging systems bring state-of-the-art technologies to open source, e.g. **DUCKDB** and **NOISEPAGE**.
- Still, much research remains proprietary / closed source, e.g. **HYPER** and **UMBRA**.
Brief Summary

• POSTGRESQL remains the top dog.

• A few emerging systems bring state-of-the-art technologies to open source, e.g. DUCKDB and NOISEPAGE.

• Still, much research remains proprietary / closed source, e.g. HYPER and UMBRA.

Subjective Problem

• It is difficult for researchers and developers to get started with the available open source projects.
Status Quo of Open Source DBMSs?

**Brief Summary**

- PostgreSQL remains the top dog.
- A few emerging systems bring state-of-the-art technologies to open source, e.g. DuckDB and NoisePage.
- Still, much research remains proprietary / closed source, e.g. HYPER and UMBRA.

**Subjective Problem**

- It is difficult for researchers and developers to get started with the available open source projects.
  - Lack of documentation.
  - Many built-in assumptions or design decisions.
A system that...
Our Vision

A system that...

• is primarily targeted at researchers and developers.
Our Vision

A system that...

• is primarily targeted at researchers and developers.
• may serve as a unifying framework for database research.
Our Vision

A system that...

• is primarily targeted at researchers and developers.
• may serve as a unifying framework for database research.
• imposes as few design decisions on the developer as possible.
Our Vision

A system that...

• is primarily targeted at researchers and developers.
• may serve as a unifying framework for database research.
• imposes as few design decisions on the developer as possible.
• is flexible and can be configured to the developer’s demands.
Our Vision

A system that...

• is primarily targeted at researchers and developers.
• may serve as a unifying framework for database research.
• imposes as few design decisions on the developer as possible.
• is flexible and can be configured to the developer’s demands.
• provides documentation for developers and eases onboarding.
Our Approach

Data Layout

Cardinality Estimation

Cost Function

Plan Enumeration

Query Execution
Our Approach

- Storage
- Data Layout

...
Our Approach

- Cardinality Estimation
- Storage
- Data Layout
Our Approach

Components

- Data Layout
- Storage
- Cardinality Estimation
- Cost Function

Plan Enumeration
Query Execution
Our Approach

Components

Cardinality Estimation

Storage

Data Layout

Cost Function

Plan Enumeration
Our Approach

Components

Cardinality Estimation

Cost Function

Plan Enumeration

Query Execution

Storage

Data Layout
Our Approach

- Data Layout
- Storage
- Cardinality Estimation
- Cost Function
- Plan Enumeration
- Query Execution
- Plan Enumeration
- ...
Our Approach

- Data Layout
- Storage
- Cardinality Estimation
- Cost Function
- Plan Enumeration
- Query Execution

...
Our Approach

- Data
- Layout
- Storage
- Cardinality Estimation
- Cost Function
- Plan Enumeration
- Query Execution
- mutable

...
Our Approach

Components

- Data Layout
- Cardinality Estimation
- Storage
- Cost Function
- Plan Enumeration
- Query Execution
- Enumeration
- Query Execution
- Plan Enumeration
- Cost Function
- Cardinality Estimation
- Storage
- Data Layout

Components
Design Goals: (1) Extensibility

- Extend mutable by implementations of a component.
- Proper documentation of components.
- Clean component API.
Design Goals: (1) Extensibility

- Extend mutable by implementations of a component.
- Proper documentation of components.
- Clean component API.

Plan Enumeration

- $DP_{size}$
- $DP_{sub}$
- $DP_{ccp}$
- $TD_{MinCutAGaT}$
- $TD_{MinCutBranch}$
- ...
Design Goals: (1) Extensibility

- Extend mutable by implementations of a component.
- Proper documentation of components.
- Clean component API.
Design Goals: (2) Separation of Concerns

- Each component is independent of other components.
- Only knowledge of API of other components may be necessary.
- In `mutable`, components must appear stateless to the outside.

- Cardinality Estimation
- Plan Enumeration
- Histograms
- Sum-Product Network
  - ...
Design Goals: (2) Separation of Concerns

- Each component is independent of other components.
- Only knowledge of API of other components may be necessary.
- In `mutable`, components must appear stateless to the outside.
Design Goals: (2) Separation of Concerns

- Each component is independent of other components.
- Only knowledge of API of other components may be necessary.
- In `mutable`, components must appear stateless to the outside.

`mutable` - Cardinality Estimation

- Histograms
- Sum-Product Network
  ...

Plan Enumeration
Design Goals: (2) Separation of Concerns

- Each component is independent of other components.
- Only knowledge of API of other components may be necessary.
- In `mutable`, components must appear stateless to the outside.

- Cardinality Estimation
- Plan Enumeration
- Histograms
- Sum-Product Network
- ...
Design Goals: (2) Separation of Concerns

- Each component is independent of other components.
- Only knowledge of API of other components may be necessary.
- In mutable, components must appear *stateless* to the outside.
Design Goals: (3) Abstraction...

```cpp
struct PlanEnumerator {
    /** Enumerate feasible plans for query \p G.
     * \param G graph representation of the query
     * \param CE cardinality estimator component of the
     *     queried database
     * \param CF cost function to minimize
     * \param PT table of best plans found, with one
     *     entry per feasible partial plan */
    virtual void enumerate_plans(
        const QueryGraph &G,      // value (in)
        const CardinalityEstimator &CE,  // component
        const CostFunction &CF,       // component
        PlanTable &PT                // value (in & out)
    ) const = 0;
};
```
Design Goals: (3) Abstraction...

```cpp
struct PlanEnumerator {
    /** Enumerate feasible plans for query \p G.
     * \param G graph representation of the query
     * \param CE cardinality estimator component of the
     * queried database
     * \param CF cost function to minimize
     * \param PT table of best plans found, with one
     * entry per feasible partial plan */
    virtual void enumerate_plans(
        const QueryGraph &G, // value (in)
        const CardinalityEstimator &CE, // component
        const CostFunction &CF, // component
        PlanTable &PT // value (in & out)
    ) const = 0;
};
```

- The PlanEnumerator component appears stateless to the outside.
Design Goals: (3) Abstraction…

```cpp
struct PlanEnumerator {
    /** Enumerate feasible plans for query \p G.
     * \param G graph representation of the query
     * \param CE cardinality estimator component of the
     *     queried database
     * \param CF cost function to minimize
     * \param PT table of best plans found, with one
     *     entry per feasible partial plan */
    virtual void enumerate_plans(
        const QueryGraph &G,    // value (in)
        const CardinalityEstimator &CE, // component
        const CostFunction &CF,     // component
        PlanTable &PT              // value (in & out)
    ) const = 0;
};
```

• The PlanEnumerator component appears stateless to the outside.
• Values go in, values come out.
Design Goals: (3) Abstraction...

```cpp
struct PlanEnumerator {
    /** Enumerate feasible plans for query \( p \) \( G \).
    * \param G graph representation of the query
    * \param CE cardinality estimator component of the queried database
    * \param CF cost function to minimize
    * \param PT table of best plans found, with one entry per feasible partial plan */
    virtual void enumerate_plans(
        const QueryGraph &G, // value (in)
        const CardinalityEstimator &CE, // component
        const CostFunction &CF, // component
        PlanTable &PT // value (in & out)
    ) const = 0;
};

• The PlanEnumerator component appears stateless to the outside.
• Values go in, values come out.
• PlanEnumerator component makes use of other components.
```
Regret?

Abstraction through dynamic dispatch (e.g. `virtual` methods) may be too much overhead for frequently called, short running functions.

- Particularly true for interpretation-based query execution.
Regret?

Abstraction through dynamic dispatch (e.g. virtual methods) may be too much overhead for frequently called, short running functions.

- Particularly true for interpretation-based query execution.

**Code Generation to the Rescue**
Regret?

Abstraction through dynamic dispatch (e.g. virtual methods) may be too much overhead for frequently called, short running functions.

• Particularly true for interpretation-based query execution.

Code Generation to the Rescue

• Use template meta programming for compile-time composition.
Regret?

Abstraction through dynamic dispatch (e.g. `virtual` methods) may be too much overhead for frequently called, short running functions.

- Particularly true for interpretation-based query execution.

**Code Generation to the Rescue**

- Use `template` meta programming for compile-time composition.
- Use code generation where the former is inapplicable.
Regret?

Abstraction through dynamic dispatch (e.g. `virtual` methods) may be too much overhead for frequently called, short running functions.

- Particularly true for interpretation-based query execution.

**Code Generation to the Rescue**

- Use `template` meta programming for compile-time composition.
- Use code generation where the former is inapplicable.
- Provide a deeply-embedded DSL, that mimics C, for easy adaption of code generation.
The Value is the Boundary

A talk by Gary Bernhardt from SCNA 2012

https://www.destroyallsoftware.com/talks/boundaries
Achieving Extensibility & Separation of Concerns

The Value is the Boundary

Cardinality Estimation
Storage
Data Layout
Cost Function
Plan Enumeration
Query Execution
...
The Value is the Boundary

- Cardinality Estimation
- Cost Function
- Plan Enumeration
- Query Execution
- Storage
- Data Layout
- mutable
  - stateful
- ...
SQL Query

\[
\ldots \text{ WHERE } x > 42 \ldots
\]
SQL Query

... WHERE x > 42 ...

Implementation of Branching Selection

IF (compile(this->condition())) {
    Pipeline();
};
Achieving Abstraction without Regret

SQL Query

... WHERE x > 42 ...

Implementation of Branching Selection

IF (compile(this->condition())) {
    Pipeline();
};

Generated WEBASSEMBLY Code

(br_if
   (i32.le_s             (; x <= 42 ;)
     (get_local $3)
     (i32.const 42)
   )
   (; Pipeline goes here ;)
)
Generated WEBASSEMBLY Code

```webassembly
(br_if
 (i32.le_s (; x <= 42 ;)
 (get_local $3)
 (i32.const 42)
)
 (; Pipeline goes here ;)
)
```
Generated WEBASSEMBLY Code

```webassembly
(br_if
 (i32.le_s  ; x <= 42 ;)
 (get_local $3)
 (i32.const 42)
)

(; Pipeline goes here ;)
)
```

- Google’s JAVASCRIPT & WEBASSEMBLY engine
- Performs JIT compilation to x86
- Tiered compilation, adaptive execution
Components Overview: Data Layout

mut**able**

Data Layout
Components Overview: Data Layout

Generic framework to express arbitrary data layouts.
Components Overview: Data Layout

**mutable**

Data Layout

- row layout
- PAX w/ varying block size

Generic framework to express arbitrary data layouts.
Components Overview: Cardinality Estimation

mutable

Cardinality Estimation

Sum-Product Networks (interpretable ML)

Histograms
Components Overview: Cost Function

### Components

- **mutable**
- **Cost Function**

---

**Cost Function**

- \( C_{\text{out}} \) by Cluet, Moerkotte
- Linear regression
- \( C_{\text{out}} \) for logical / algebraic join ordering
- Linear regression trained with automatic benchmarks on physical operators, used for selecting physical operators
Components Overview: Cost Function

- $C_{out}$ for logical / algebraic join ordering
- linear regression trained with automatic benchmarks on physical operators, used for selecting phys. operators
Components Overview: Plan Enumeration

\[\text{mutable} \rightarrow \text{Plan Enumeration}\]
In this work, we investigate join order optimization via heuristic search. In particular, we make the following contributions.

1. To the best of our knowledge, we present the first formal framing of the join order optimization problem (JOOP) and argue why and how heuristic search is applicable to join order optimization.
2. We define heuristic search, perform a theoretical analysis of its capabilities for JOOP, and elaborate the general search procedure.
3. We present an efficient search space representation for both bottom-up and top-down join ordering and elaboration.
4. We identify and circumvent a potential pitfall when incorporating a state-of-the-art cost model into heuristic search, that severely burdens a DBMS with determining a query execution plan.
5. We experimentally evaluate our approach and compare it to two state-of-the-art approaches.
6. We propose a new benchmark that systematically explores the Query Graph Exploration Landscape (QGraEL) along two crucial optimizations, one of which is highly particular to recently investigated machine learning.

The join order has a major impact on the performance of the query. To compute without exhaustively exploring the entire search space. It has proven itself efficiently finding optimal or nearly optimal solutions without exhaustively to explore exhaustively. DBMSs therefore define a threshold beyond which the amount of plans is too large to consider. In particular, we make the following contributions.

In this work, we present a new approach to join order optimization that is based on heuristic search, an important class of such algorithms is algorithms to efficiently explore such search spaces. An important topic of research. The scope of algorithms to compute join orders ranges from exhaustive enumeration, to combinatorics, to greedy search, to genetic algorithms, based on graph properties, to heuristic search. In particular, we make the following contributions.

In the domain of AI planning, searching extremely large search spaces is a frequent task and research in that area has brought forth many algorithms.

The join order has a major impact on the performance of the query. To compute without exhaustively exploring the entire search space. It has proven itself efficiently finding optimal or nearly optimal solutions without exhaustively to explore exhaustively. DBMSs therefore define a threshold beyond which the amount of plans is too large to consider. In particular, we make the following contributions.

In this work, we present a new approach to join order optimization that is based on heuristic search, an important class of such algorithms is algorithms to efficiently explore such search spaces. An important topic of research. The scope of algorithms to compute join orders ranges from exhaustive enumeration, to combinatorics, to greedy search, to genetic algorithms, based on graph properties, to heuristic search. In particular, we make the following contributions.

In the domain of AI planning, searching extremely large search spaces is a frequent task and research in that area has brought forth many algorithms.
**Efficiently Computing Join Orders with Heuristic Search**

Immanuel Haffner  
Saarland Informatics Campus  
immanuel.haffner@bigdata.uni-saarland.de

Jens Dittrich  
Saarland Informatics Campus  
jens.dittrich@bigdata.uni-saarland.de

**ABSTRACT**

Join order optimization is one of the most fundamental problems in processing queries on relational data. It has been studied extensively for almost four decades now. Still, because of its NP hardness, no generally efficient solution exists and the problem remains an important topic of research. The scope of algorithms to compute join orders ranges from exhaustive enumeration, to combinatorics based on graph properties, to greedy search, to genetic algorithms, to recently investigated machine learning. A few works exist that use heuristic search to compute join orders. However, a theoretical understanding of the applicability of heuristic search is missing.

Contributions.

In this work, we present a new approach to join order optimization that is based on heuristic search, an important class of such algorithms. We formally state the join order optimization problem (JOOP) and show that it is related to a well-known problem of finding the minimum cut in a graph. This allows us to use heuristic search applied to our shortest path problem, and devise crucial lemmata for both bottom-up and top-down join ordering and reduction of JOOP to shortest path. We present formalizations for both bottom-up and top-down join ordering and reductions for both bottom-up and top-down search. Additionally, we devise a theoretical framework, and the applicability of heuristic search. We devise crucial lemmata for both bottom-up and top-down join ordering and reduction of JOOP to shortest path. We present formalizations for both bottom-up and top-down search. Additionally, we devise a theoretical framework, and the applicability of heuristic search.

We then thoroughly analyze the properties of this problem. We then thoroughly analyze the properties of this problem. We thoroughly analyze the properties of this problem. We thoroughly analyze the properties of this problem.

Up to 1000x faster than state of the art (DP_{ccp})
In this paper, we argue that we should design compiling query
interpreters as well. As interpreters, they transform SQL queries to
machine code and do not include any optimization.

Recently, Kohn et al. proposed an adaptive approach to query
processing that switches from non-optimized to optimized code
during query processing. This approach requires immense
development effort, ultimately preventing widespread adoption.

The reason for an extensive body of work on query interpretation
is that implementing either approach requires expertise in inter-
preting SQL queries to executable code. However, recent works
as well as mutable (ours) that use an architecturally much simpler
architecture as part of a real database system: mutable (ours)
demonstrates how to implement this architecture in a real
database system: mutable (ours)

Our results show that we are able to match or even
speed up query execution. The QEP is then executed by either interpretation
time or compilation. Many early database systems used an interpreter
rather than being reinvented by database architects. By carefully
choosing a suitable code compilation and execution engine we
are able to get just-in-time code compilation (including the full
rewritten code) and compilation time.

In the past decade, we have witnessed considerable progress in
query compilation. Query compilation is crucial to efficiently execute query plans.

In this work, we propose a new architecture for query exe-
cution that defines how to execute SQL queries. We provide an extensive experimental study, showcasing
that we are able to reduce compilation times while maintaining competitive
execution time. We demonstrate how to implement this architecture in a real
database system: mutable (ours)

We discuss in detail the pros and cons over compiling with
WebAssembly
and compiler design and poses an immense development
effort, ultimately preventing widespread adoption.

We propose this conceptual architecture using
the compiler construction community, like reg-
eration. This approach is not viable in the long run – we argue that code compilation and execu-
tion engines conceptually very differently: rather than racing against
the different engines of database systems. Rather than reengineering
these approaches require immense engineering effort, a consid-
erable part of which includes reengineering very fundamental
query execution plan (QEP) that defines how to exe-
cute SQL queries.

We claim that we can get just-in-time (JIT) compilation, tiered compilation, and adaptive execution to an under-
lying engine. Like that, we avoid reengineering techniques
那一刻, 我们需要考虑一个实际的数据库系统: mutable (ours)

We discuss current limitations of our approach and how they
will get resolved in the (near) future.

As an overall conclusion, we claim that we will get just-in-time (JIT) compilation, tiered compilation, and adaptive execution to an under-
lying engine. Like that, we avoid reengineering techniques
那一刻, 我们需要考虑一个实际的数据库系统: mutable (ours)

We claim that we can get just-in-time (JIT) compilation, tiered compilation, and adaptive execution to an under-
lying engine. Like that, we avoid reengineering techniques
那一刻, 我们需要考虑一个实际的数据库系统: mutable (ours)

We discuss current limitations of our approach and how they
will get resolved in the (near) future.
Components Overview: Query Execution

mutable

Query Execution

Interpreter (discont’d)
A Simplified Architecture for Fast, Adaptive Compilation and Execution of SQL Queries

Immanuel Haffner
Saarland Informatics Campus
immanuel.haffner@bigdata.uni-saarland.de

Jens Dittrich
Saarland Informatics Campus
jens.dittrich@bigdata.uni-saarland.de

Figure 1: Design space of query execution engines, based on TPC-H Q1 benchmark results. The compilation time is the time to translate a QEP to machine code. The execution time is the time to execute the machine code and does not include the compilation time.

JIT compilation, tiered compilation, and adaptive execution

adopted by many database systems that followed [15]. The induced overhead of interpretation was dwarfed by the high costs for data accesses in disk-based systems [8, 22, 32]. However, in modern main memory systems data accesses are significantly faster and the interpretation overhead suddenly takes a large share in query execution costs [3, 32]. Therefore, main memory systems must keep any overheads during query execution at a minimum to achieve peak performance. This development was the reason for an extensive body of work on query interpretation
Continuous Benchmarking

- automated benchmarking (nightly)
- automatic detection of performance anomalies
Questions?

github.com/mutable-org/mutable
Backup Slides
Components Overview

- **Data Layouts:**
  - generic framework, arbitrary layouts; implemented row, PAX with varying block size

- **Cardinality Estimation:**
  - Sum-Product Networks (interpretable ML), Histograms

- **Cost Functions:**
  - for algebraic/logical optimization: $C_{\text{out}}$ by Cluet, Moerkotte
  - for physical optimization: linear regression trained on autom. benchmarks

- **Plan Enumeration:**
  - $\text{DP}_{\text{size}}, \text{DP}_{\text{sub}}, \text{DP}_{\text{ccp}}, \text{TD}_{\text{MinCutAGaT}}$
  - our Heuristic Search, published @SIGMOD'23 (up to 1000x faster than $\text{DP}_{\text{ccp}}$)

- **Query Execution:**
  - query compilation to WEBASSEMBLY, tiered compilation & adaptive execution in Google's V8, published @EDBT'23 (similar UMBRA's Tidy Tuples & Flying Start)
A Simplified Architecture for Fast, Adaptive Compilation and Execution of SQL Queries

Immanuel Haffner
Saarland Informatics Campus
immanuel.haffner@bigdata.uni-saarland.de

Jens Dittrich
Saarland Informatics Campus
jens.dittrich@bigdata.uni-saarland.de

Efficiently Computing Join Orders with Heuristic Search

Immanuel Haffner
Saarland Informatics Campus
immanuel.haffner@bigdata.uni-saarland.de

Jens Dittrich
Saarland Informatics Campus
jens.dittrich@bigdata.uni-saarland.de

ABSTRACT

Query compilation is a key component of database system performance, and a popular method to improve query execution is to aggressively switch from non-optimized to optimized code. However, the drawbacks of this approach are well-known: the difficulties in integrating optimization studies efficiently with the runtime of a query execution system. We propose a novel architecture that allows us to omit these complications and to fully leverage the power of modern compiler construction engines.

In this paper, we describe a new architecture for query execution that avoids the need for integrated compiler compilation and runtime optimization techniques. Rather than reengineering the existing components of a database system, our architecture abstracts away the optimization process and relies on a state-of-the-art compilation and runtime engine. This allows us to focus on the end-to-end query execution process, which we optimize using a novel compiler construction approach.

We experimentally evaluate our approach and compare it to state-of-the-art architectures. In our experiments, we observe a significant performance improvement over existing systems, which is achieved by exploiting the power of modern compiler engines to aggressively optimize query execution.

Efficiently Computing Join Orders with Heuristic Search

We consider the problem of computing join orders, which is a critical step in the query optimization process. Our approach is based on heuristic search, which is known to be effective in finding good solutions in a reasonable amount of time. We demonstrate the effectiveness of our method by comparing it to several state-of-the-art approaches.

In this paper, we describe a new approach to computing join orders that is based on heuristic search. We evaluate our approach using a variety of benchmark queries and show that it is able to find good join orders in a fraction of the time required by existing methods.

ABSTRACT

Join order optimization is one of the most fundamental problems in processing queries on relational data. It has been studied extensively for almost four decades. Still, because of its NP hardness, no generally efficient solution exists and the problem remains an important topic of research. The scope of algorithms to compute join orders ranges from exhaustive enumeration, to combinatorics based on graph properties, to greedy search, to genetic algorithms, to recently investigated machine learning. A few works exist that use heuristic search to compute join orders. However, a theoretical argument why and how heuristic search is applicable to join order optimization is lacking.

In this work, we investigate join order optimization via heuristic search. In particular, we provide a strong theoretical framework, required by a query are done. A crucial part of determining a query plan is determining a join order, i.e. the order in which individual relations are joined by the respective join predicates of the query. The join order has a major impact on the performance of the query plan and hence it is of utmost importance to a DBMS to compute a “good” join order – or at least to avoid “bad” join orders [2, 19]. This problem is known as the join order optimization problem (JOOP) and it is generally NP hard [4, 16]. There exists a comprehensive body of work on computing join orders. It can be divided into work on computing optimal join orders [4, 7, 12, 13, 16, 22, 32], work on greedy computation of potentially suboptimal join orders [10, 24, 25, 37], work on adaptive re-optimization of join orders [17, 26, 28, 38], and recent work based on machine learning [20, 21, 23].

Ono and Lohman [27] derive analytically the number of distinct