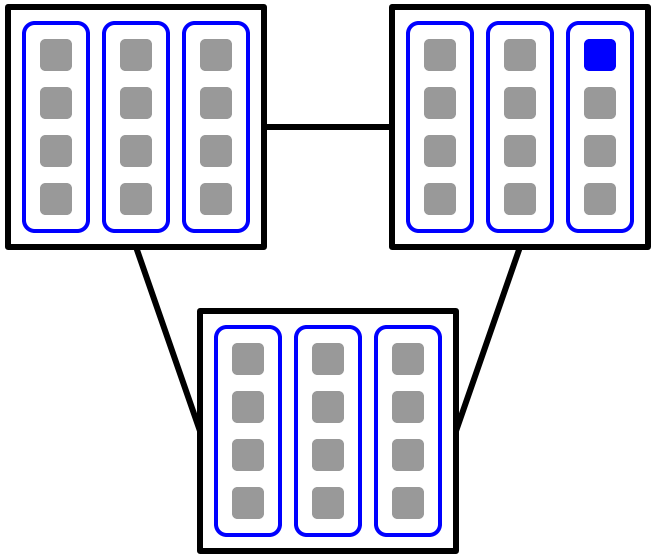
Как мы проектировали распределенный SQL-движок в Hazelcast

Владимир Озеров

Что такое Hazelcast IMDG?

- Распределенное in-memory key-value хранилище

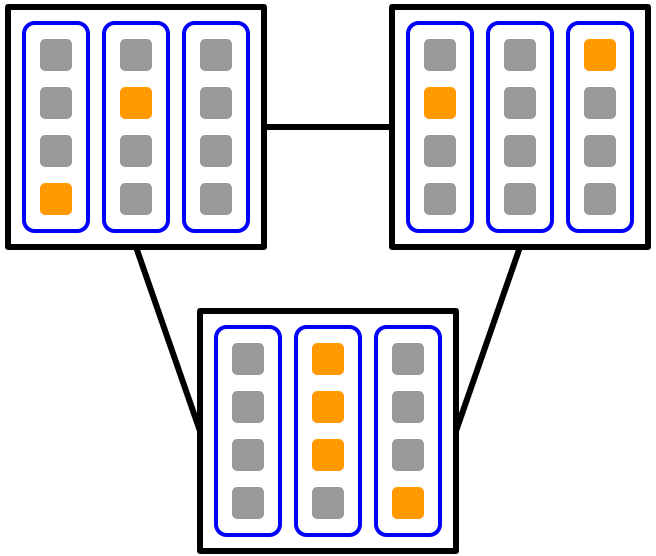


```
IMap<Long, Person> map = …
...
map.put(1L, new Person(“John”));
...
Person person = map.get(1L);
```


hazelcast

Что не так с key-value API?

- Доступ **только** по ключу
- Хотим делать более сложный анализ: поиск по предикату, агрегации, joins, ...



```
IMap<Long, Person> map = …
...
Result r = map.find([name = "John"]);
```

Что у нас было?

- **Predicate API** - возможность найти данные по предикату
- In-memory **индексы** для ускорения поиска по предикату
  - Heap: конкурентный skip-list
  - Offheap: самописное однопоточное красно-чёрное дерево

```
IMap<Long, Person> map = …
...
map.put(1L, new Person("John"));
...
Predicate predicate = Predicates.equals("name", "John");
Collection<Person> persons = map.values(predicate);
```

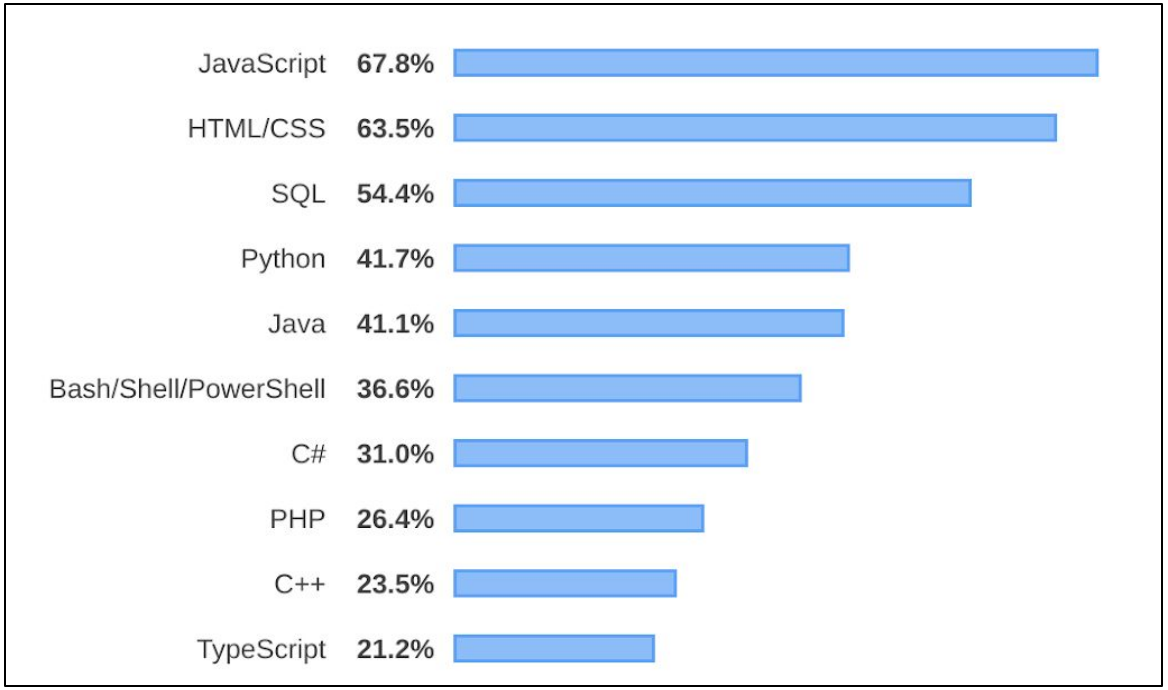
Что не так с Predicate API?

- Работает с множествами (`Collection`), может упасть с OOME
- Ограниченный функционал (например, как сделать join?)
- Необходимо разбираться с документацией
- Необходимо компилировать

```
IMap<Long, Person> map = …
...
map.put(1L, new Person("John"));
...
Predicate predicate = Predicates.equals("name", "John");
Collection<Person> persons = map.values(predicate);
```

Решение: SQL!

- Популярен!
- Декларативен: не надо ничего компилировать
- Не привязан к Java: доступ с разных платформ
- Экосистема: утилиты и интеграции на любой вкус



hazelcast

Что мы сделали в релизе 4.1?

Функционально

- Поддержка распределенных запросов `SELECT … FROM … WHERE`
- Поддержка индексов

Технически

- Оптимизатор запросов на основе Apache Calcite
- Протокол распределенного выполнения запросов
- Неблокирующая кооперативная модель параллельной обработки
- Основа для будущих улучшений (join/sort/aggregate, компиляция, ...)

Что мы сделали в релизе 4.1?

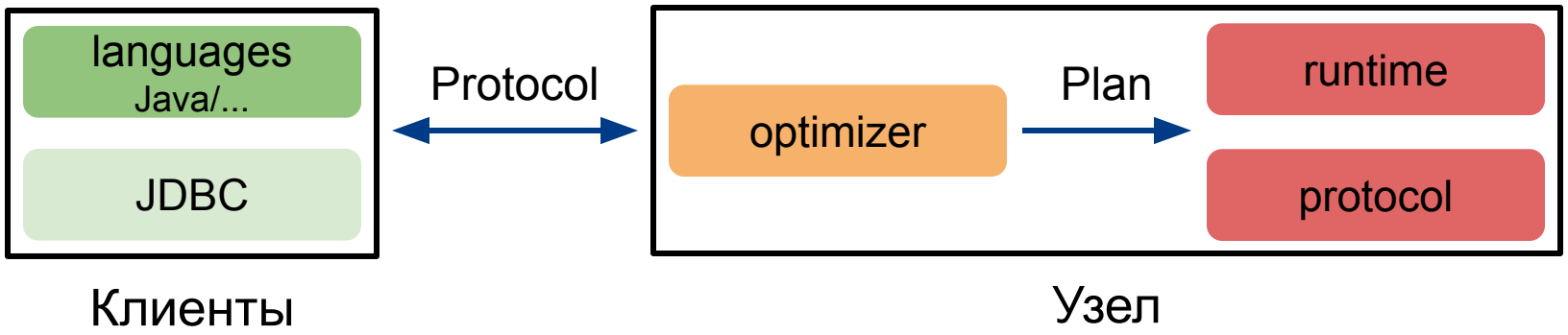
```
IMap<Long, Person> map = instance.getMap("person");

...

try (SqlResult result = instance.getSql().query("SELECT id, name FROM person") {
    for (SqlRow row : result) {
        System.out.println("Name: " + row.getObject("name"));
    }
}
```

- Интуитивный API:
  - Напоминает JDBC, но проще
  - Работаем с итераторами, а не коллекциями
- Отсутствие конфигурации:
  - Автоматически определяем структуру данных

Архитектура



hazelcast

О чем доклад?

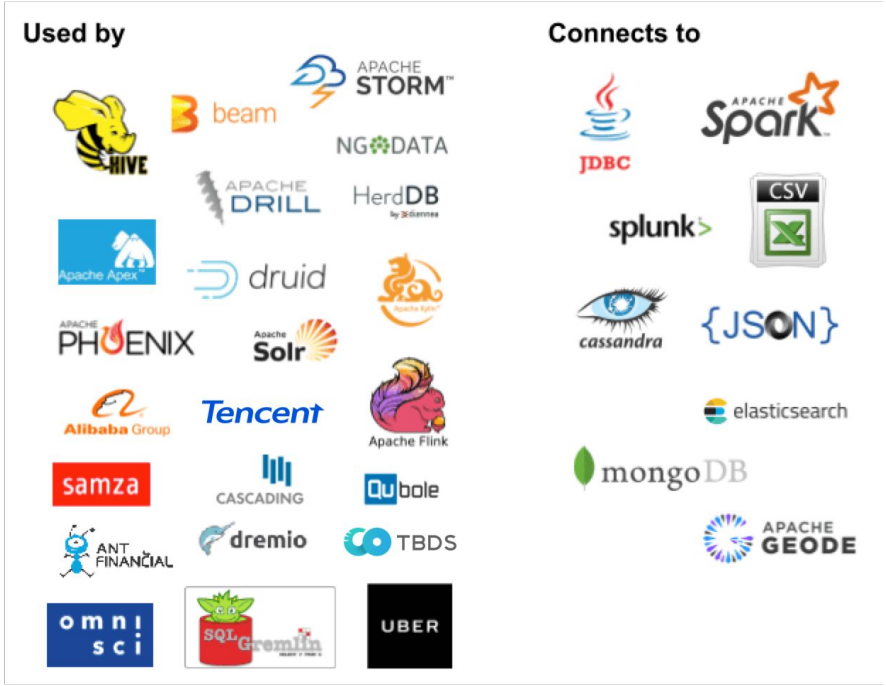
- Какие технические решения принимали, и почему
  - Оптимизатор запросов
  - Локальный runtime
  - Сетевой протокол

hazelcast

Оптимизатор

Предварительный анализ

- **Индустрия**: “оптимизатор это **сложно**”
- **Практика**: многие Java-проекты используют Apache Calcite [1]
- **Идея**: начнем с Apache Calcite



[1] https://calcite.apache.org/docs/powered_by.html

hazelcast

Возможности Apache Calcite

Apache Calcite - “dynamic data management **framework**”

- Парсер ANSI SQL
- Логический оптимизатор
- Runtime (компиляция, коннекторы к другим системам)
- JDBC

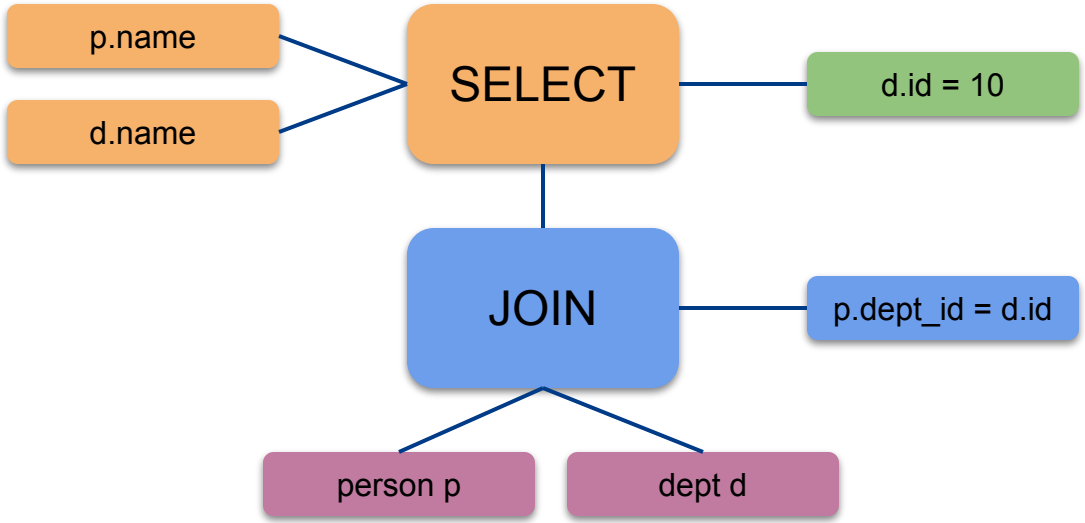
Процесс оптимизации

Синтаксический анализ

- **Задача**: проверка синтаксиса
- **Вход**: строка
- **Выход**: синтаксическое дерево (aka AST)

Синтаксическое дерево

```
SELECT p.name, d.name
FROM person p JOIN dept d
  ON p.dept_id = d.id
WHERE d.id = 10
```



hazelcast

Синтаксический анализ

Рассуждения:

- Задача понятная, но объемная
- Писать руками - долго! (пример: H2 [1])
- Использовать parser generator - все равно долго! (пример: Postgres [2])
- Apache Calcite имеет готовый парсер, который легко расширять [3]

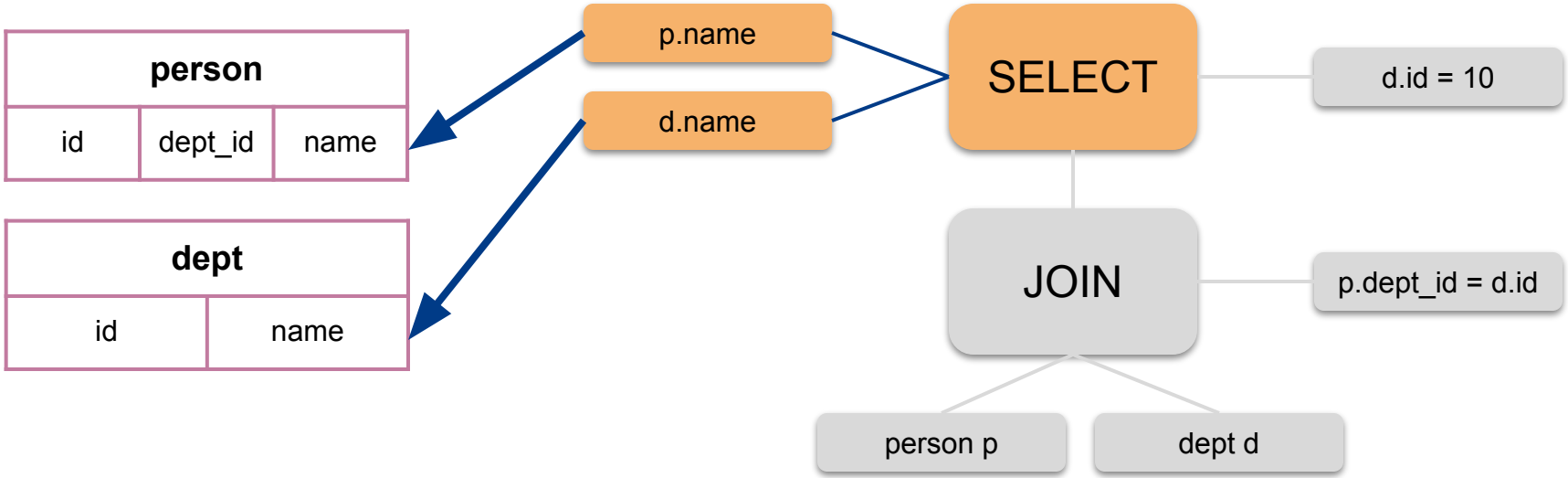
Решение: парсим через Apache Calcite

[1] https://github.com/h2database/h2database/blob/master/h2/src/main/org/h2/command/Parser.java
[2] https://github.com/postgres/postgres/tree/master/src/backend/parser
[3] https://github.com/apache/calcite/blob/master/core/src/main/codegen/templates/Parser.jj

Семантический анализ

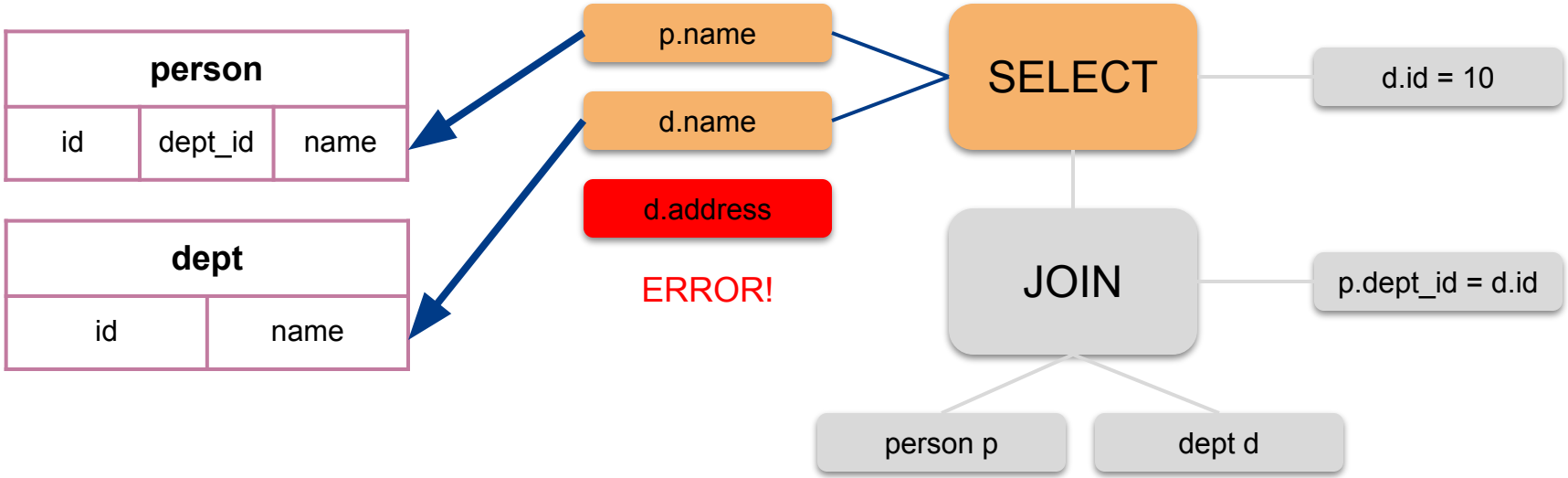
- **Задача**:
  - Проверка логической корректности запроса
  - Трансформация синтаксического дерева в реляционное дерево (опционально)
- **Вход**: синтаксическое дерево
- **Выход**: логическое реляционное дерево

Семантический анализ: корректность



hazelcast

Семантический анализ: корректность



hazelcast

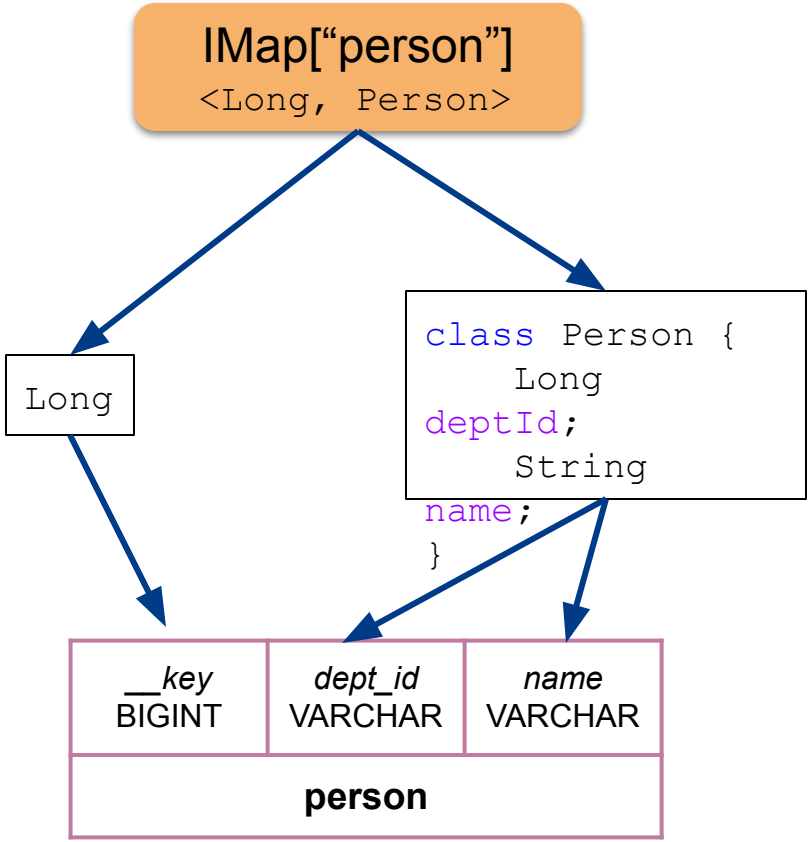
Семантический анализ: схема

Проблема:

- SQL предполагает наличие схемы (таблицы, атрибуты)
- В Hazelcast нет понятия схемы, мы храним произвольные объекты

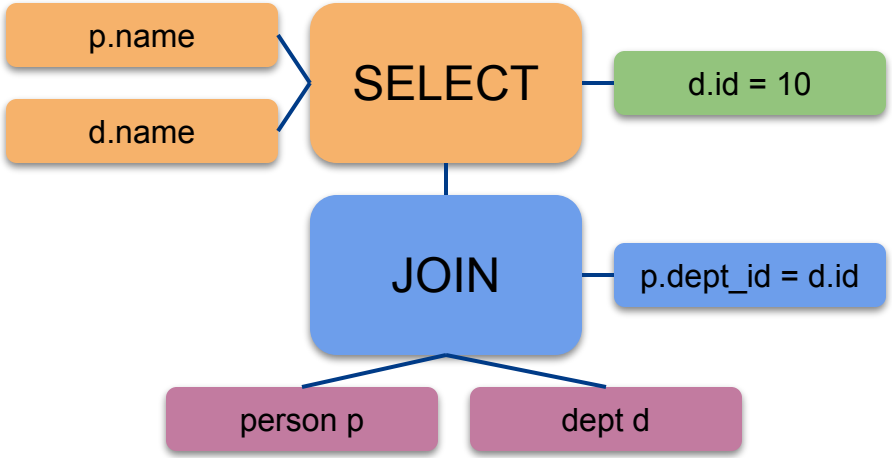
Решение:

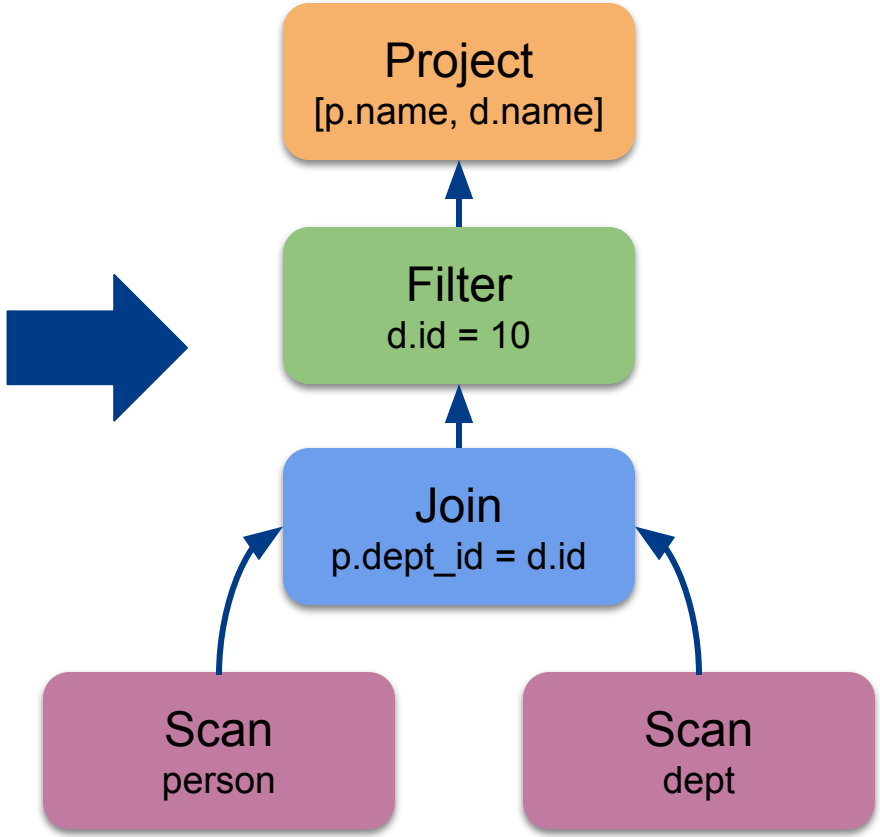
- Представляем каждый `IMap` как таблицу
- Находим первую key-value пару в `IMap`, извлекаем из нее атрибуты интроспекцией (например, reflection)
- Во время исполнения запроса: если последующие пары имеют другие типы, бросаем ошибку



hazelcast

Дерево реляционных операторов





hazelcast

Семантический анализ: решение

Рассуждения:

- Задача не очень понятная, и объемная
- Писать руками - долго!
- Apache Calcite имеет готовый семантический анализатор, но!
  - Код: сложный, монолитный, проблемы с абстракциями и расширяемостью, баги
  - Поведение, которое нас не устраивает в некоторых случаях (например, приведение типов)

Решение:

- Используем Apache Calcite [1] [2]
- Переписываем часть кода, вставляем “костыли”

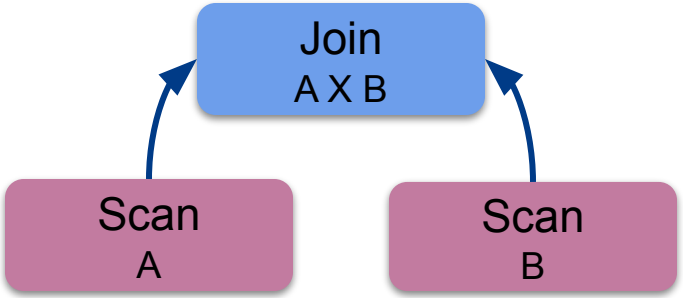
[1] https://github.com/apache/calcite/blob/master/core/src/main/java/org/apache/calcite/sql/validate/SqlValidatorImpl.java
[2] https://github.com/apache/calcite/blob/master/core/src/main/java/org/apache/calcite/sql2rel/SqlToRelConverter.java

Оптимизация

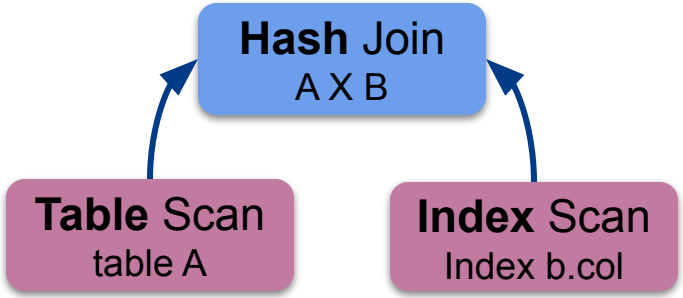
- **Задача**: найти наиболее дешевый физический план
- **Вход**: логическое реляционное дерево
- **Выход**: оптимизированное физическое реляционное дерево

Логические и физические операторы

Логическое дерево



Физическое дерево



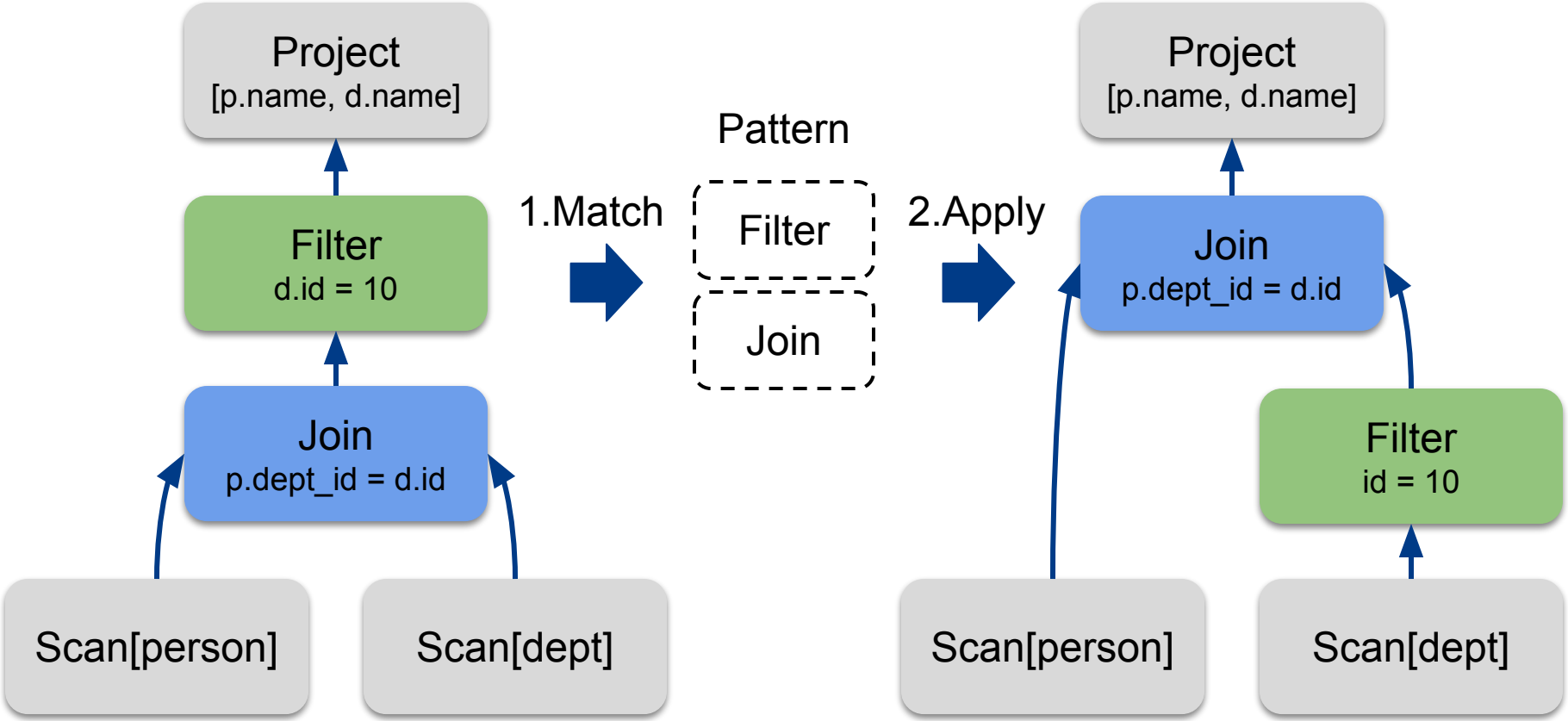
- **Логические операторы** - “что делать?” (напр, join)
- **Физические операторы** - “как делать?” (напр, **hash** join)
- Одному **логическому** оператору соответствует один или несколько **физических**

hazelcast

Логические и физические операторы

Логический оператор	Физические операторы
Scan	Table Scan, Index Scan
Aggregate	Hash Aggregate, Streaming Aggregate
Join	Hash Join, Merge Join, Nested Loop Join

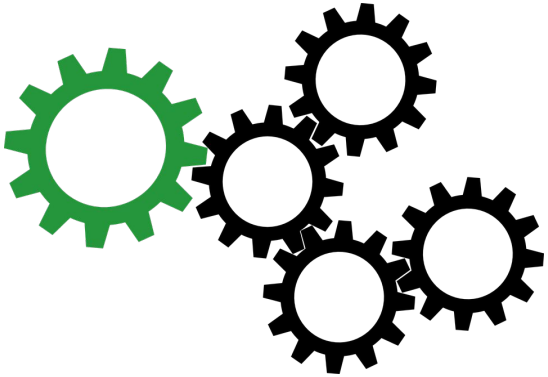
Правила



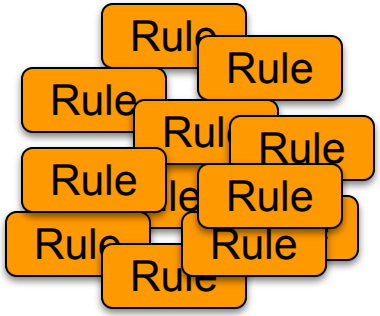
hazelcast

Принцип

Логический план



Физический план



Логика

Метаданные

hazelcast

Принцип: “System R” vs “Cascades”

Access Path Selection in a Relational Database Management System

P. Griffiths Selinger
M. M. Astrahan
D. D. Chamberlin
R. A. Lorie
T. G. Price

IBM Research Division, San Jose, California 95193

ABSTRACT: In a high level query and data manipulation language such as SQL, requests are stated non-procedurally, without reference to access paths. This paper describes how System R chooses access paths for both simple (single relation) and complex queries (such as joins), given a user specification of desired data as a boolean expression of predicates. System R is an experimental database management system developed to carry out research on the relational model of data. System R was designed and built by members of the IBM San Jose Research Laboratory.

1. Introduction

System R is an experimental database management system based on the relational model of data which has been under development at the IBM San Jose Research Laboratory since 1975 <1>. The software was developed as a research vehicle in relational database, and is not generally available outside the IBM Research Division.

This paper assumes familiarity with relational data model terminology as described in Codd <7> and Date <8>. The user interface in System R is the unified query, data definition, and manipulation language SQL <5>. Statements in SQL can be issued both from an on-line casual-user-oriented terminal interface and from programming languages such as PL/I and COBOL.

In System R a user need not know how the tuples are physically stored and what access paths are available (e.g. which columns have indexes). SQL statements do not require the user to specify anything about the access path to be used for tuple retrieval. Nor does a user specify in what order joins are to be performed. The System R optimizer chooses both join order and an access path for each table in the SQL statement. Of the many possible choices, the optimizer chooses the one which minimizes "total access cost" for performing the entire statement.

This paper will address the issues of access path selection for queries. Retrieval for data manipulation (UPDATE, DELETE) is treated similarly. Section 2 will describe the place of the optimizer in the processing of a SQL statement, and section 3 will describe the storage component access paths that are available on a single physically stored table. In section 4 the optimizer cost formulas are introduced for single table queries, and section 5 discusses the joining of two or more tables, and their corresponding costs. Nested queries (queries in predicates) are covered in section 6.

2. Processing of an SQL statement

A SQL statement is subjected to four phases of processing. Depending on the origin and contents of the statement, these phases may be separated by arbitrary intervals of time. In System R these arbitrary time intervals are transparent to the system components which process a SQL statement. These mechanisms and a description of the processing of SQL statements from both programs and terminals are further discussed in <2>. Only an overview of those processing steps that are relevant to access path selection will be discussed here.

The four phases of statement processing are parsing, optimization, code generation, and execution. Each SQL statement is sent to the parser, where it is checked for correct syntax. A query block is represented by a SELECT list, a FROM list, and a WHERE tree, containing, respectively the list of items to be retrieved, the table(s) referenced, and the boolean combination of simple predicates specified by the user. A single SQL statement may have many query blocks because a predicate may have one operand which is itself a query.

If the parser returns without any errors detected, the OPTIMIZER component is called. The OPTIMIZER accumulates the names

Originally published in the *Proceedings of the 1979 ACM SIGMOD International Conference on the Management of Data.*

Digital recreation by Eric A. Brewer, brewer@cs.berkeley.edu, October 2002.

The Cascades Framework for Query Optimization

Goetz Graefe

Abstract

This paper describes a new extensible query optimization framework that resolves many of the shortcomings of the EXODUS and Volcano optimizer generators. In addition to extensibility, dynamic programming, and memorization based on and extended from the EXODUS and Volcano prototypes, this new optimizer provides (i) manipulation of operator arguments using rules or functions, (ii) operators that are both logical and physical for predicates etc., (iii) schema-specific rules for materialized views, (iv) rules to insert "enforcers" or "glue operators," (v) rule-specific guidance, permitting grouping of rules, (vi) basic facilities that will later permit parallel search, partially ordered cost measures, and dynamic plans, (vii) extensive tracing support, and (viii) a clean interface and implementation making full use of the abstraction mechanisms of C++. We describe and justify our design choices for each of these issues. The optimizer system described here is operational and will serve as the foundation for new query optimizers in Tandem's NonStop SQL product and in Microsoft's SQL Server product.

1 Introduction

Following our experiences with the EXODUS Optimizer Generator [GrD87], we built a new optimizer generator as part of the Volcano project [GrM93]. The main contributions of the EXODUS work were the optimizer generator architecture based on code generation from declarative rules, logical and physical algebra's, the division of a query optimizer into modular components, and interface definitions for support functions to be provided by the database implementor (DBI), whereas the Volcano work combined improved extensibility with an efficient search engine based on dynamic programming and memorization. By using the Volcano Optimizer Generator in two applications, a object-oriented database systems [BMG93] and a scientific database system prototype [WoG93], we identified a number of flaws in its design. Overcoming these flaws is the goal of a completely new extensible optimizer developed in the Cascades project, a new project applying many of the lessons learned from the Volcano project on extensible query optimization, parallel query execution, and physical database design. Compared to the Volcano design and implementation, the new Cascades optimizer has the following advantages. In their entirety, they represent a substantial improvement over our own earlier work as well as other related work in functionality, ease-of-use, and robustness.

- Abstract interface classes defining the DBI-optimizer interface and permitting DBI-defined subclass hierarchies
- Rules as objects
- Facilities for schema- and even query-specific rules
- Simple rules requiring minimal DBI support
- Rules with substitutes consisting of a complex expression

hazelcast

Оптимизация: решение

Рассуждения:

- Написать алгоритм оптимизации - сложно, но можно попробовать
- Но нужны еще и правила - это объемная работа
- Apache Calcite:
  - Имеет готовые планировщики (heuristic [1], cost-based [2])
  - Имеет **богатую** библиотеку правил трансформации [3] и метаданных [4]
  - Легко расширяем
  - Cost-based планировщик не масштабируется!
  - Не имеет правил имплементации (напр. выбор индекса)

Решение:

- Используем Apache Calcite для оптимизации
- Дописываем свои правила [5]

[1] https://github.com/apache/calcite/tree/master/core/src/main/java/org/apache/calcite/plan/hep
[2] https://github.com/apache/calcite/tree/master/core/src/main/java/org/apache/calcite/plan/volcano
[3] https://github.com/apache/calcite/tree/master/core/src/main/java/org/apache/calcite/rel/rules
[4] https://github.com/apache/calcite/tree/master/core/src/main/java/org/apache/calcite/rel/metadata

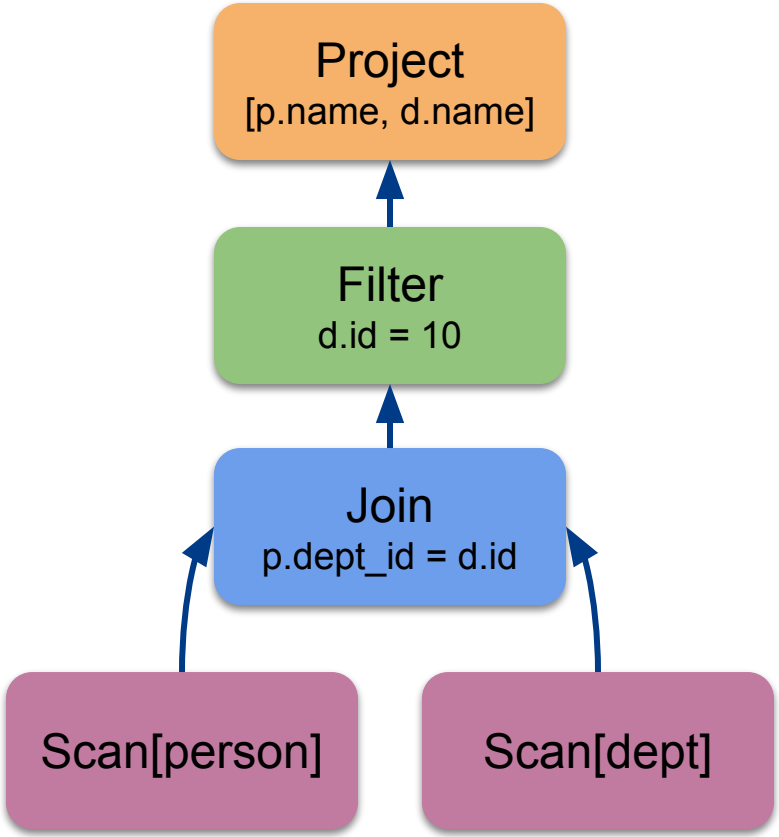
Transformation: правила

- Упрощение выражений (напр. constant folding)
- Перемещение операторов (напр. filter pushdown)
- Разворачивание подзапросов (subquery unnesting [1])
- Объединение операторов (operator merge/fusion)
- Удаление ненужных операторов (напр. `”GROUP BY unique_column -> no-op”`)
- … тысячи их [2]

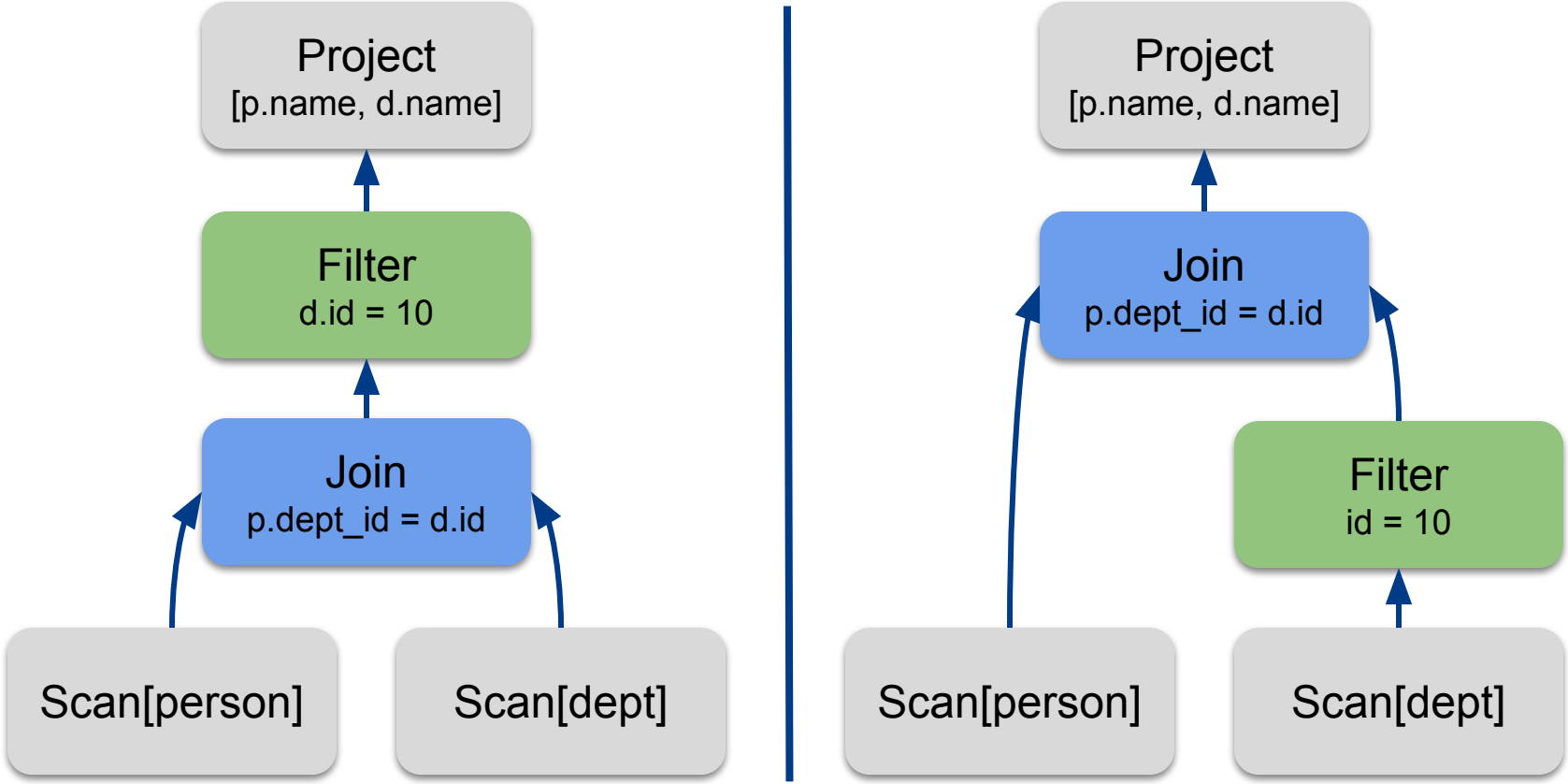
- Большинство transformation правил работают с логическими операторами, поэтому их применение иногда называют “logical optimization”

[1] https://www.semanticscholar.org/paper/Unnesting-Arbitrary-Queries-Neumann-Kemper/3112928019f64d8c388e8cfbae34b9887c789213
[2] https://github.com/apache/calcite/tree/master/core/src/main/java/org/apache/calcite/rel/rules

Transformation

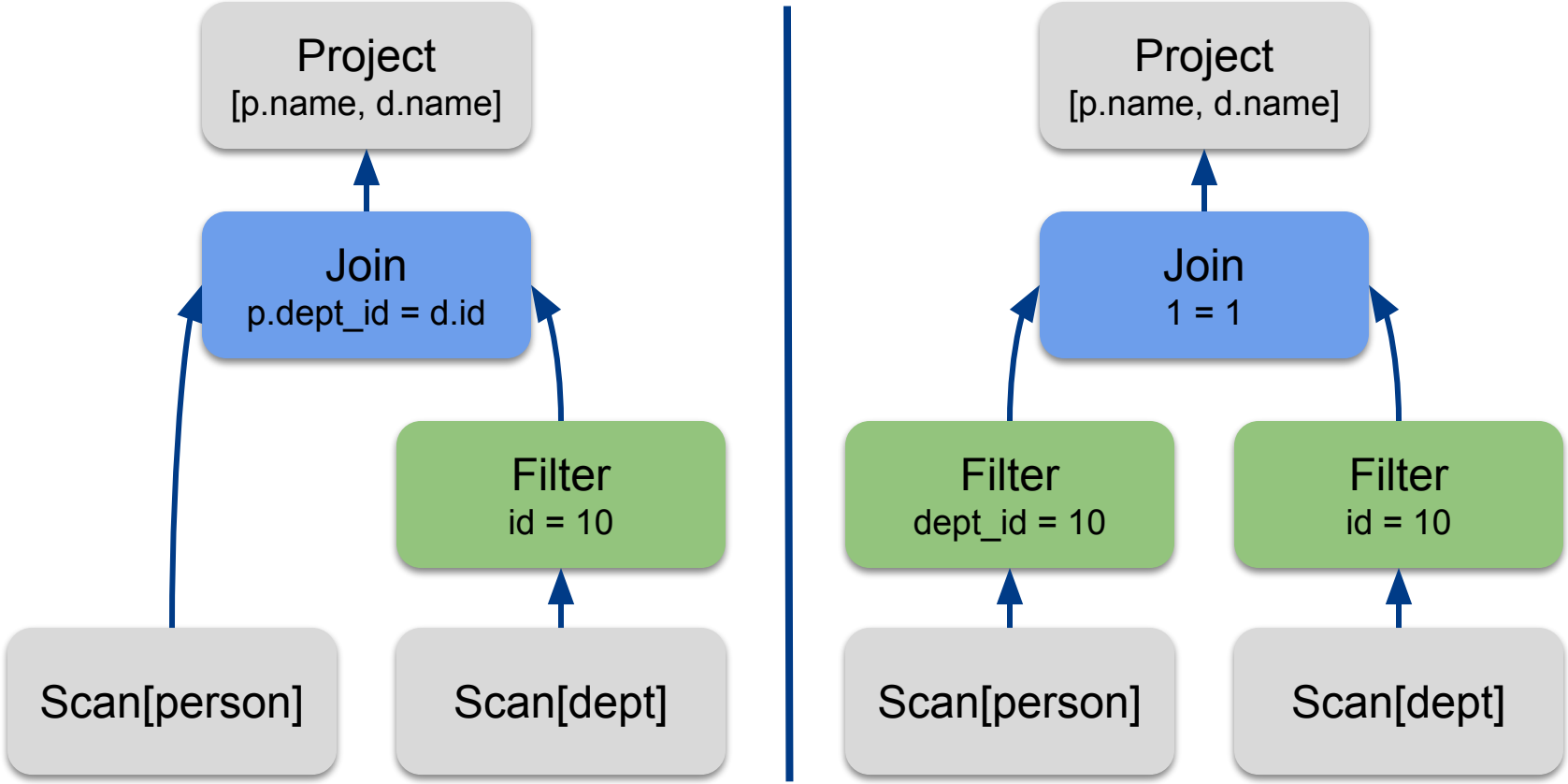


Transformation: filter push-down



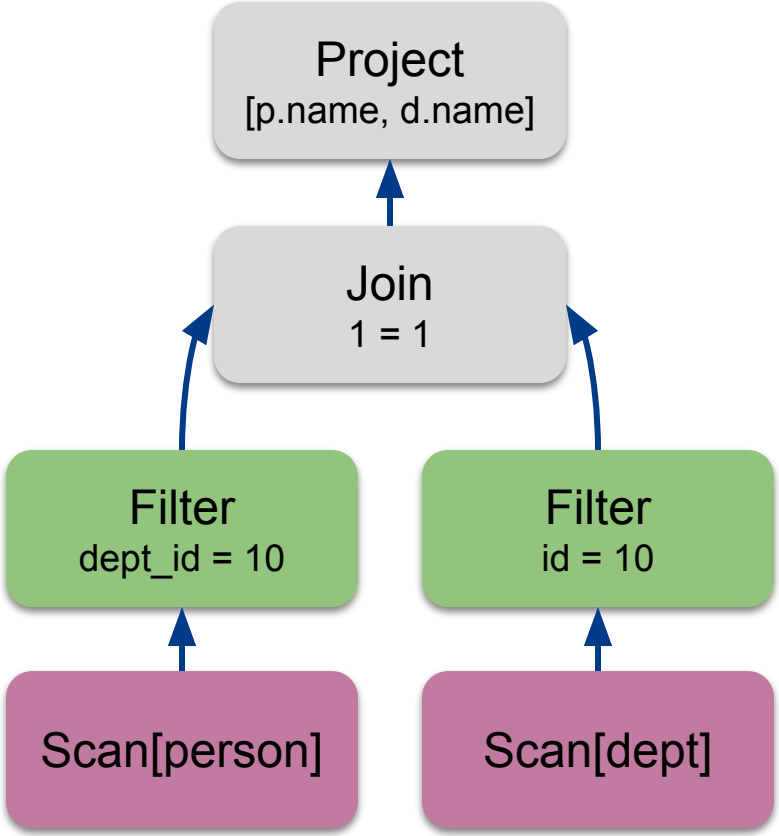
hazelcast

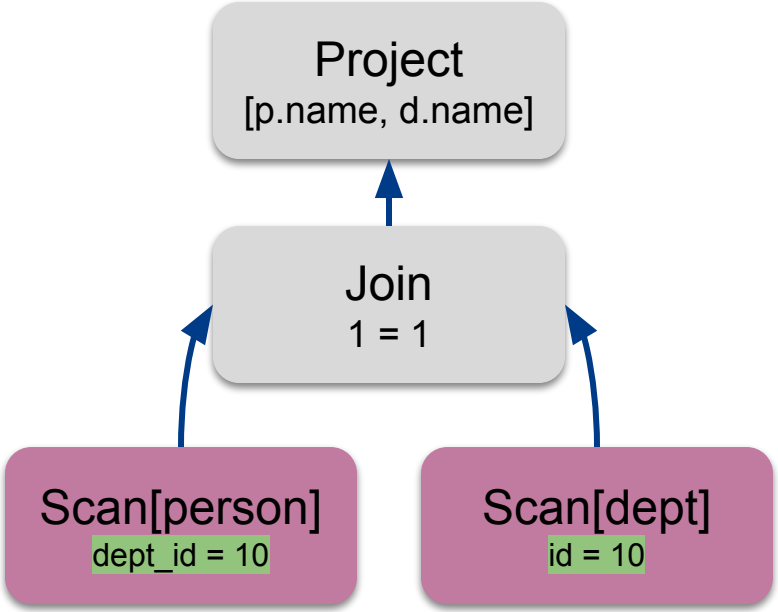
Transformation: filter move-around



hazelcast

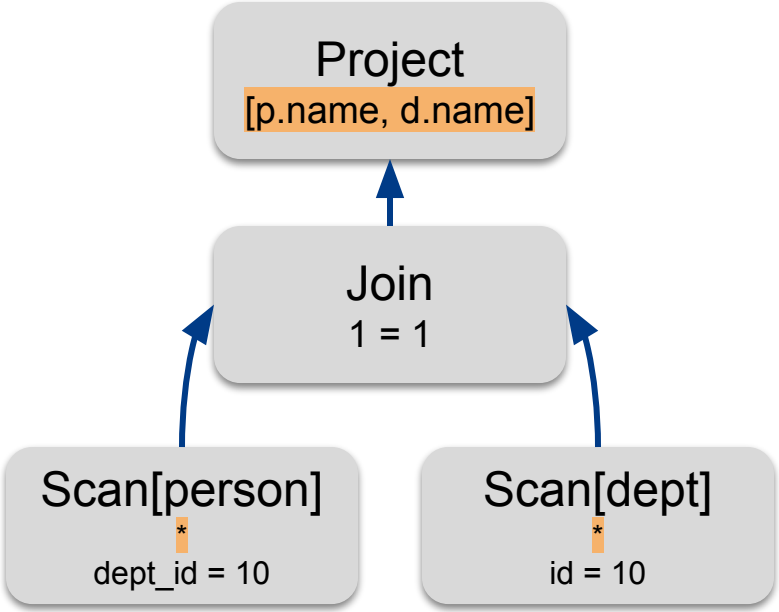
Transformation: constrained scan (filter)

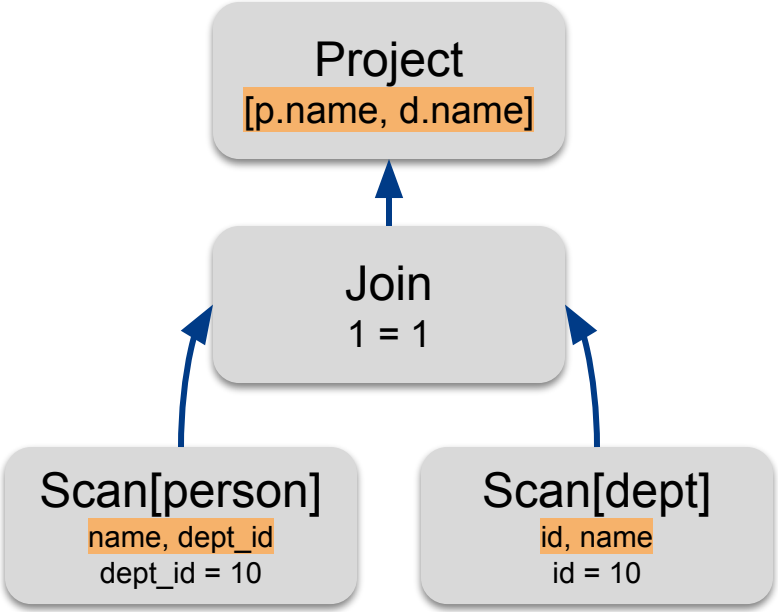




hazelcast

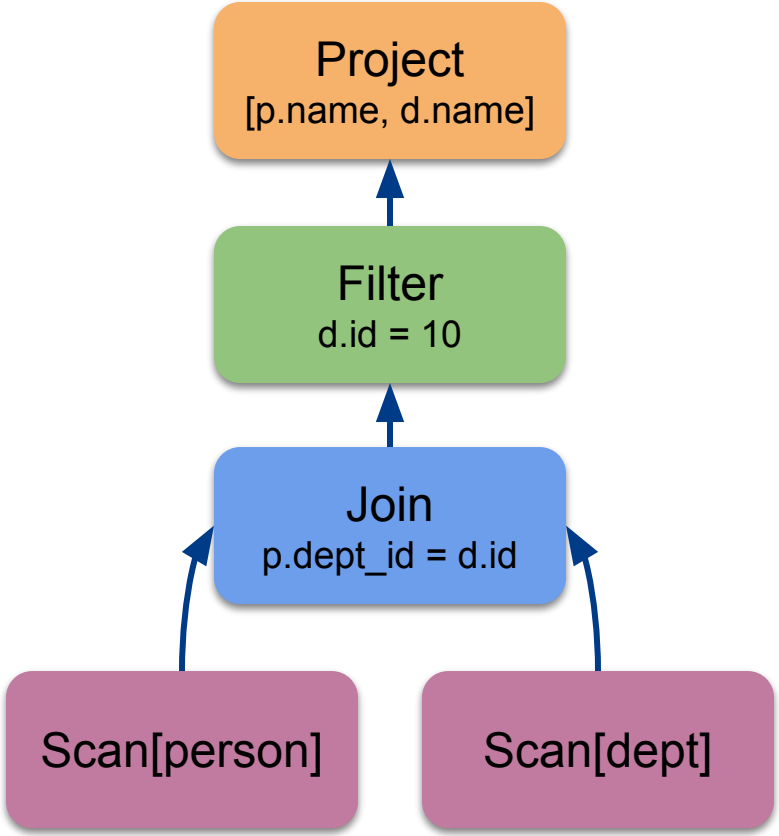
Transformation: constrained scan (project)

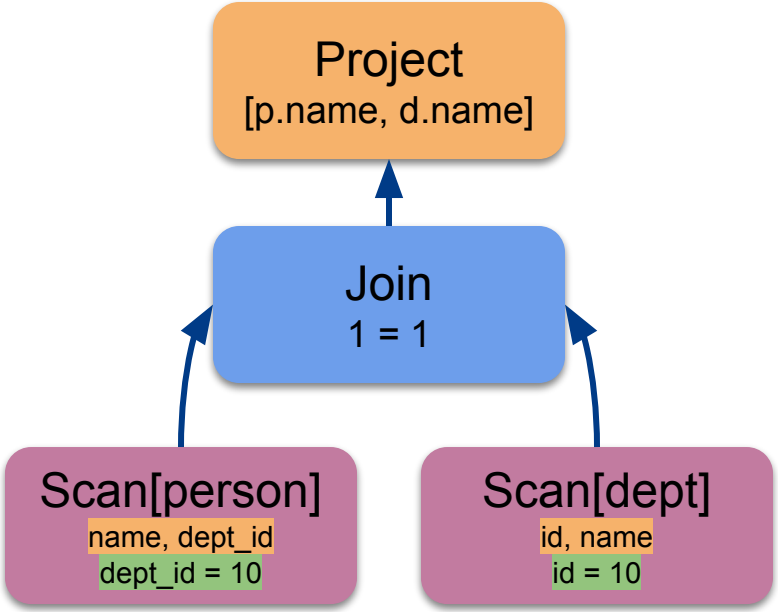




hazelcast

Transformation: результат





hazelcast

Implementation: правила

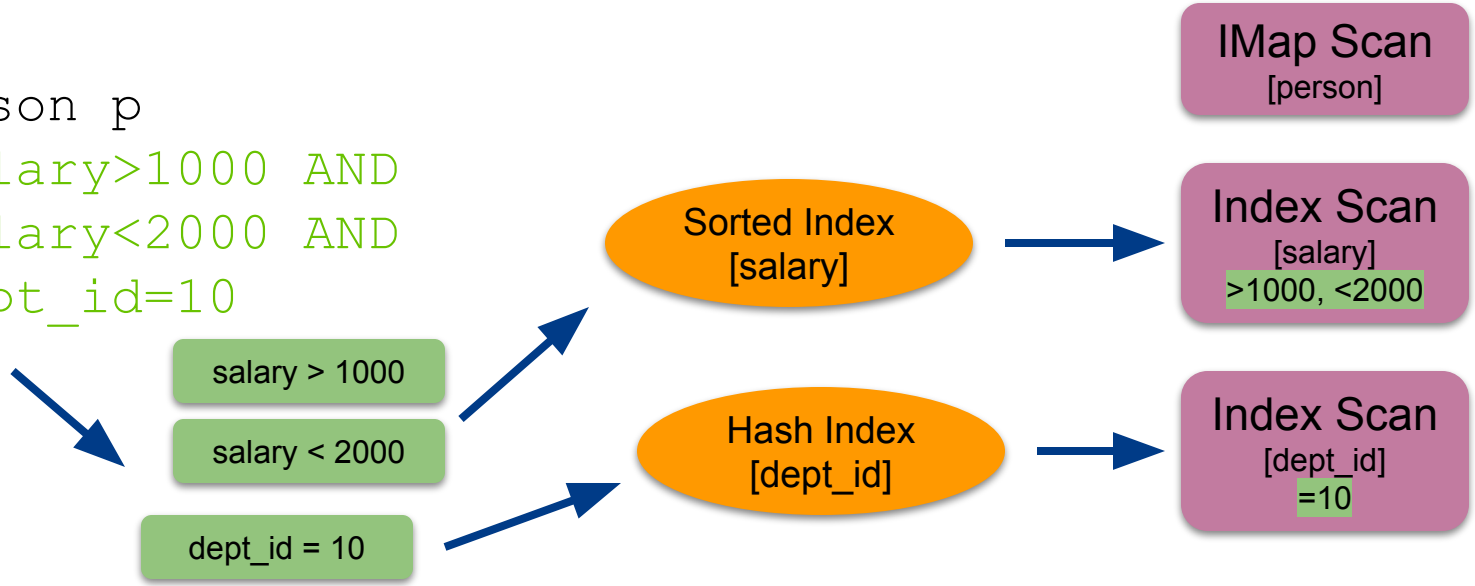
- **Готовые**
  - Выбор индексов [1]
  - Обмен данных между узлами [2]

- **Прототипы**
  - Двухфазная распределенная сортировка
  - Двухфазный распределенный aggregate
  - Колокация данных для JOIN

[1] https://github.com/hazelcast/hazelcast/blob/master/hazelcast-sql/src/main/java/com/hazelcast/sql/impl/calcite/opt/physical/MapScanPhysicalRule.java
[2] https://github.com/hazelcast/hazelcast/blob/master/hazelcast-sql/src/main/java/com/hazelcast/sql/impl/calcite/opt/distribution/DistributionTraitDef.java

Выбор индексов

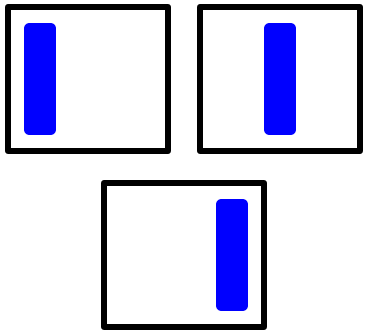
```
SELECT *
FROM person p
WHERE salary>1000 AND
      salary<2000 AND
      dept_id=10
```



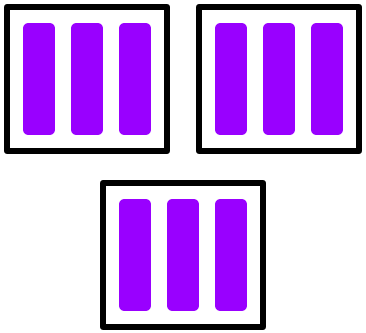
- Разбиваем конъюнкцию на отдельные предикаты
- Находим подходящие индексы
- Добавляем [Index Scan] в оптимизатор

hazelcast

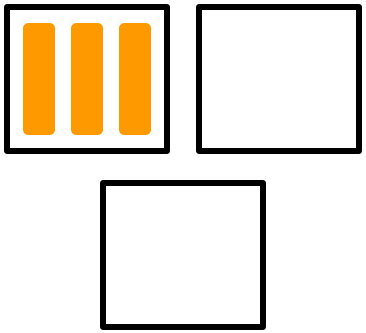
Обмен данных между узлами



PARTITIONED
Данные распределены по узлам
(напр. `IMap`)



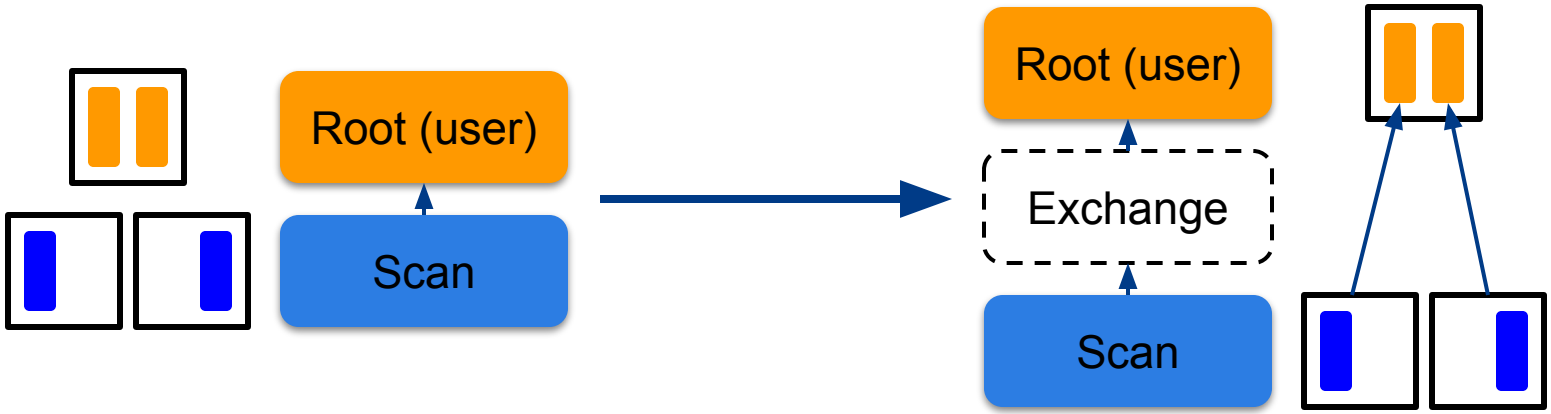
REPLICATED
Каждый узел имеет полную копию данных
(напр. `ReplicatedMap`)



SINGLETON
Данные находятся на одном узле
(напр. курсор пользователя)

- Каждый оператор имеет свойство **distribution**, которое описывает, как данные распределены в кластере
- Оптимизатор задает распределение текущего оператора на основе метаданных, или распределений дочерних операторов

Обмен данных между узлами



- Если распределение соседних операторов несовместимо, вставляем специальный оператор **Exchange**, который пересылает данные по сети

hazelcast

Итого

- **Синтаксический анализ**
  - Apache Calcite: парсер
  - Hazelcast: расширения парсера
- **Семантический анализ**
  - Apache Calcite: база (с опаской)
  - Hazelcast: множество наших расширений
- **Оптимизация**
  - Apache Calcite: планировщик (с опаской), transformation rules, метаданные
  - Hazelcast: implementation rules, метаданные

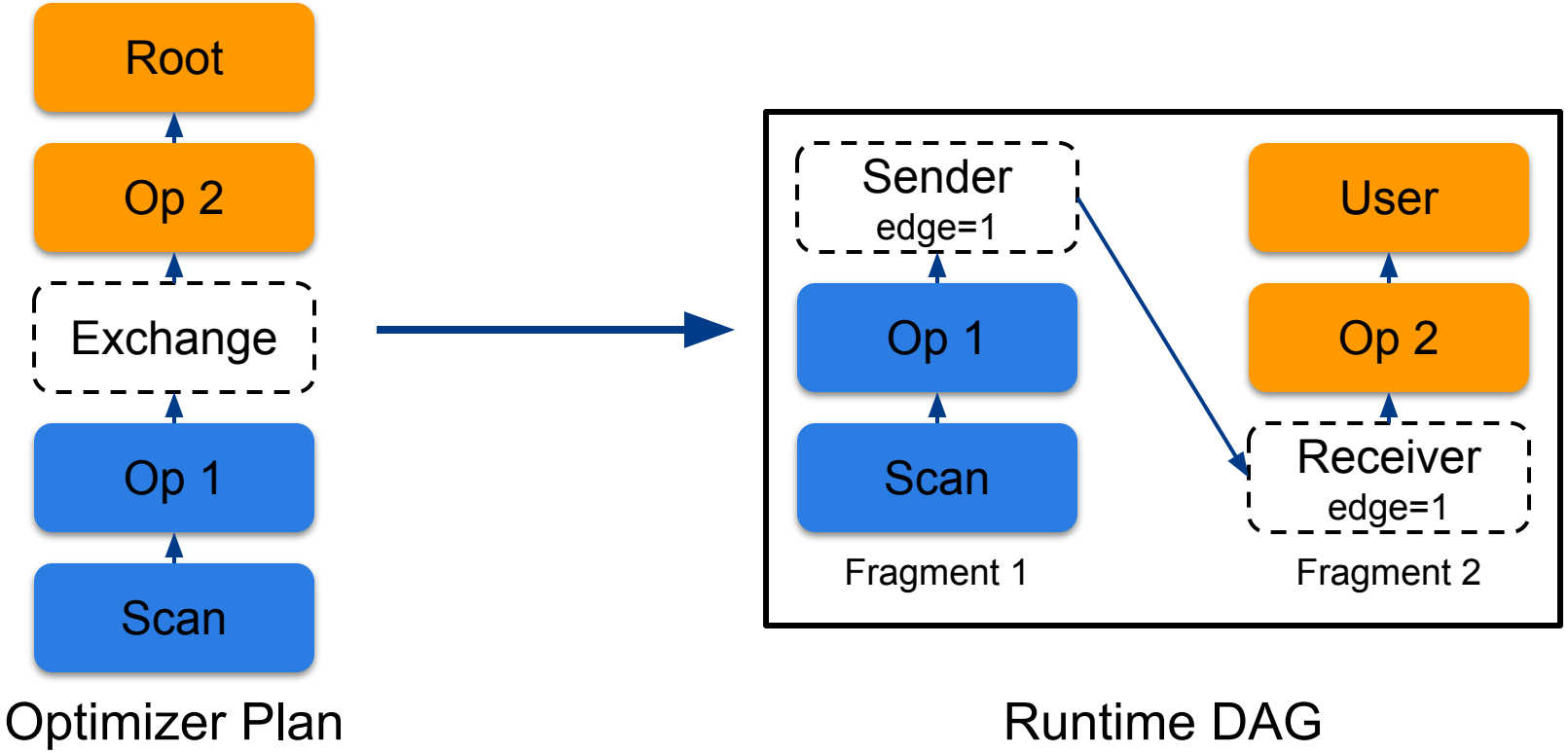
hazelcast

Runtime

Создание плана: DAG

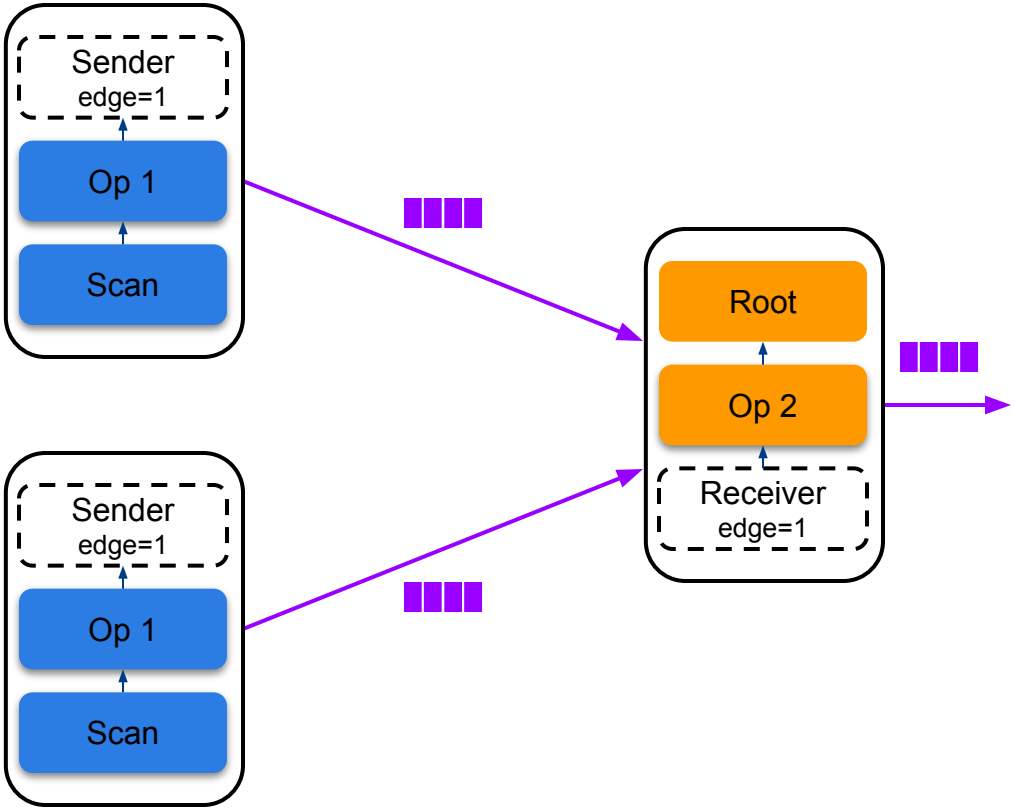
- **Sender** - оператор, который отправляет данные
- **Receiver** - оператор, который принимает данные
- **Fragment -** дерево операторов, которое может быть исполнено на одном узле, независимо от других узлов и фрагментов

Создание плана: DAG



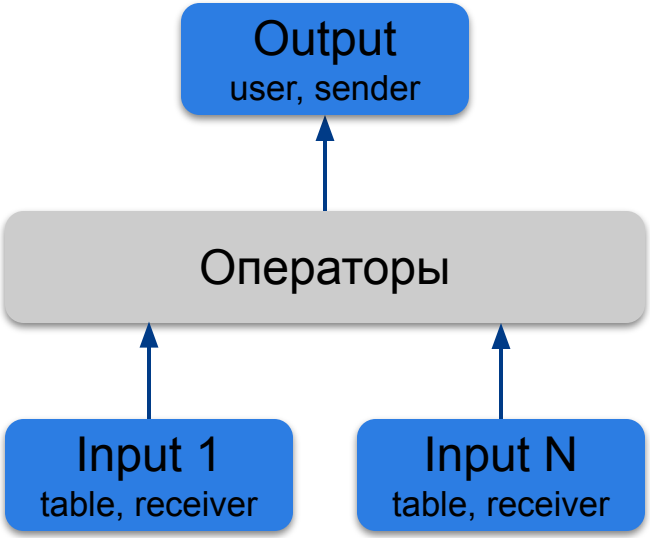
hazelcast

Создание плана: DAG



hazelcast

Структура фрагмента



- Один или несколько input-ов (таблица, сеть)
- Один output (пользователь, сеть)
- Произвольное количество промежуточных операторов

Volcano Model

```
interface Exec {
    void open();
    Row next();
    void close();
}
```

120 IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. 6, NO. 1, FEBRUARY 1994

Volcano—An Extensible and Parallel Query Evaluation System

Goetz Graefe

***Abstract*—To investigate the interactions of extensibility and parallelism in database query processing, we have developed a new dataflow query execution system called Volcano. The Volcano effort provides a rich environment for research and education in database systems design, heuristics for query optimization, parallel query execution, and resource allocation.**

Volcano uses a standard interface between algebra operators, allowing easy addition of new operators and operator implementations. Operations on individual items, e.g., predicates, are imported into the query processing operators using *support functions*. The semantics of support functions is not prescribed; any data type including complex objects and any operation can be realized. Thus, Volcano is *extensible* with new operators, algorithms, data types, and type-specific methods.

Volcano includes two novel *meta-operators*. The *choose-plan* meta-operator supports *dynamic query evaluation plans* that allow delaying selected optimization decisions until run-time, e.g., for embedded queries with free variables. The *exchange* meta-operator supports *intra-operator parallelism* on partitioned datasets and both *vertical and horizontal inter-operator parallelism*, translating between demand-driven dataflow within processes and data-driven dataflow between processes.

All operators, with the exception of the exchange operator, have been designed and implemented in a single-process environment, and parallelized using the exchange operator. Even operators not yet designed can be parallelized using this new operator if they use and provide the interator interface. Thus, the issues of data manipulation and parallelism have become *orthogonal*, making Volcano the first implemented query execution engine that effectively combines extensibility and parallelism.

***Index Terms*—Dynamic query evaluation plans, extensible database systems, iterators, operator model of parallelization, query execution.**

I. Introduction

IN ORDER to investigate the interactions of extensibility, efficiency, and parallelism in database query processing and to provide a testbed for databse systems research and education, we have designed and implemented a new query evaluation system called Volcano. It is intended to provide an experimental vehicle for research into query execution techniques and query optimization optimization heuristics rather than a database system ready to support applications. It is not a complete database system as it lacks features such as a user-friendly query language, a type system for instances (record definitions), a query optimizer, and catalogs. Because of this focus, Volcano is able to serve as an experimental vehicle for a multitude of purposes, all of them open-ended, which results in a combination of requirements that have not been integrated in a single system before. First, it is modular and extensible to enable future research, e.g., on algorithms, data models, resource allocation, parallel execution, load balancing, and query optimization heuristics. Thus, Volcano provides an infrastructure for experimental research rather than a final research prototype in itself. Second, it is simple in its design to allow student use and research. Modularity and simplicity are very important for this purpose because they allow students to begin working on projects without an understanding of the entire design and all its details, and they permit several concurrent student projects. Third, Volcano's design does not presume any particular data model; the only assumption is that query processing is based on transforming sets of items using parameterized operators. To achieve data model independence, the design very consistently separates set processing *control* (which is provided and inherent in the Volcano operators) from *interpretation and manipulation* of data items (which is imported into the operators, as described later). Fourth, to free algorithm design, implementation, debugging, tuning, and initial experimentation from the intricacies of parallelism but to allow experimentation with parallel query processing. Volcano can be used as a single-process or as a parallel system. Single-process query evaluation plans can already be parallelized easily on shared-memory machines and soon also on distributed-memory machines. Fifth, Volcano is realistic in its query execution paradigm to ensure that students learn how query processing is really done in commercial database products. For example, using temporary files to transfer data from one operation to the next as suggested in most textbooks has a substantial performance penalty, and is therefore used in neither real database systems nor in Volcano. Finally, Volcano's means for parallel query processing could not be based on existing models since all models explored to date have been designed with a particular data model and operator set in mind. Instead, our design goal was to make parallelism and data manipulation *orthogonal*, which means that the mechanisms for parallel query processing are independent of the operator set and semantics, and that all operators, including new

Manuscript received July 26, 1990; revised September 5, 1991. This work was supported in part by the National Science Foundation under Grants IRI-8996270, IRI-8912618, and IRI-9006348, and by the Oregon Advanced Computing Institute (OACIS).

The author is with the Computer Science Department, Portland State University, Portland, OR 97207-0751.

IEEE Log Number 9211308.

hazelcast

Volcano Model: пример

```
interface Exec {
    void open();
    Row next();
    void close();
}
```

```
class FilterExec implements Exec {

    Exec input;
    Expression filter;

    Row next() {
        while (true) {
            Row row = input.next();

            if (filter.eval(row))
                return row;
        }
    }
}
```

Volcano и буферы

Block oriented processing of Relational Database operations in modern Computer Architectures (2001)

```
interface Exec {
    void open();
    Row next();
    void close();
}
```

```
interface Exec {
    void open();
    List<Row> next();
    void close();
}
```

+ Амортизирует накладные расходы на row
+ Открывает новые оптимизации (например, векторизацию)
– Все еще является блокирующим!
– Требует больше памяти
– Запрограммировать запрос руками все равно гораздо эффективнее

Volcano без блокировок

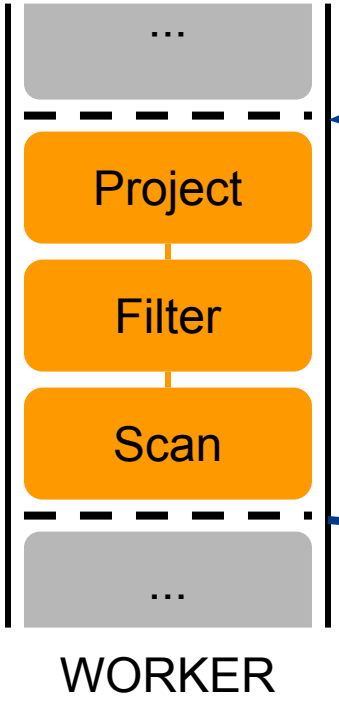
```
interface Exec {
    void open();
    List<Row> next();
    void close();
}
```

```
interface Exec {
    void open();
    IterationResult next();
    List<Row> currentBatch();
    void close();
}

enum IterationResult {
    FETCHED,
    FETCHED_DONE,
    WAIT
}
```

- Если оператор не может продолжить работу, он возвращает **WAIT**
- Родительские операторы “прокидывают” **WAIT** вверх по стеку, и “отпускают” поток

Запуск фрагмента



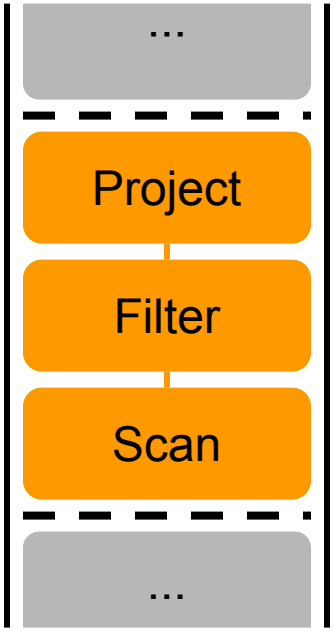

```
void schedule() {
    if (scheduled.cas(false, true))
        pool.submit(this);
}
```

```
void unschedule() {
    scheduled.set(false);

    if (!batches.isEmpty())
        schedule();
}
```


hazelcast

Запуск фрагмента

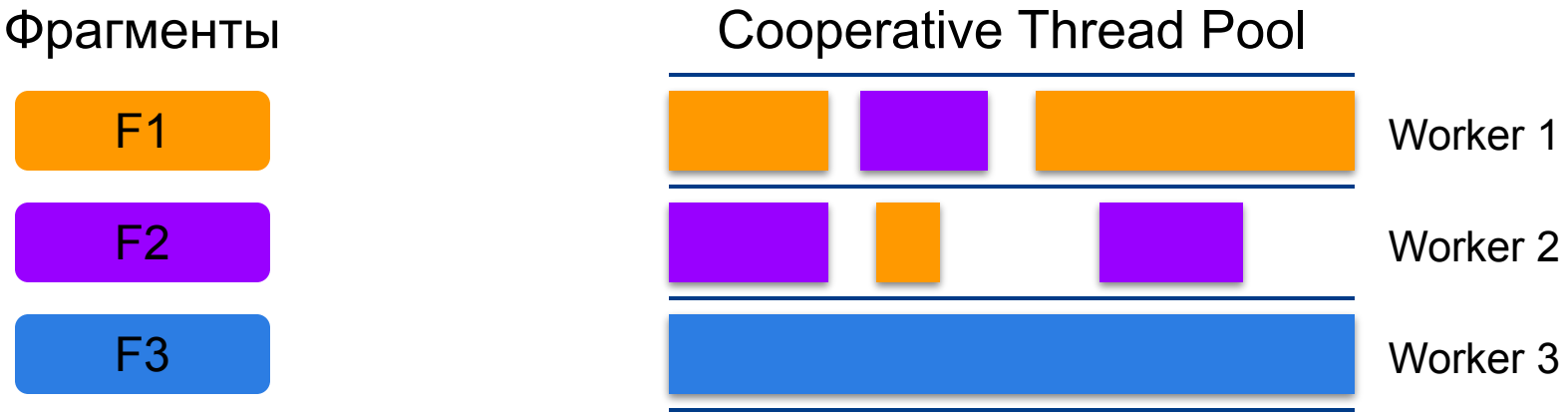


Все операторы являются **однопоточными!**

- Простота реализации
- Высокая производительность из-за отсутствия concurrency
- Нет горизонтального масштабирования!

hazelcast

Кооперативность



- Если фрагмент не может продолжать работу, он выгружается, а не блокирует поток

hazelcast

Thread Pool

Требования

- Балансировка: задачи распределяются между потоками
- Производительность: чем выше, тем лучше

`ThreadPoolExecutor`

- Балансировка: да, общая очередь
- Производительность: одна блокирующая очередь

`ForkJoinPool`

- Балансировка: да, много очередей + work stealing
- Производительность: превосходит `ThreadPoolExecutor` на маленьких запросах

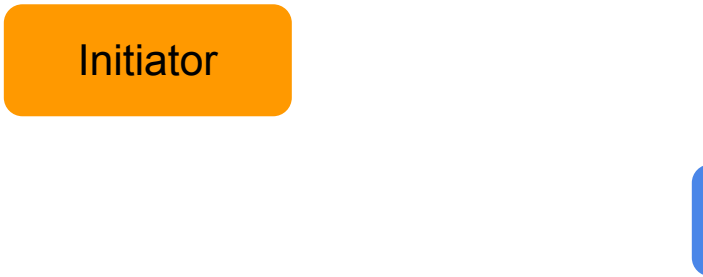
Итого

- Распределенный запрос состоит из 1 или нескольких фрагментов
- **Фрагмент** - дерево операторов, выполняемых локально
- Фрагменты связаны через **Sender/Receiver**
- Текущая модель выполнения: неблокирующий **pull** с батчингом
- Идеи на будущее:
  - управление памятью
  - компиляция
  - горизонтальный параллелизм

hazelcast

Протокол

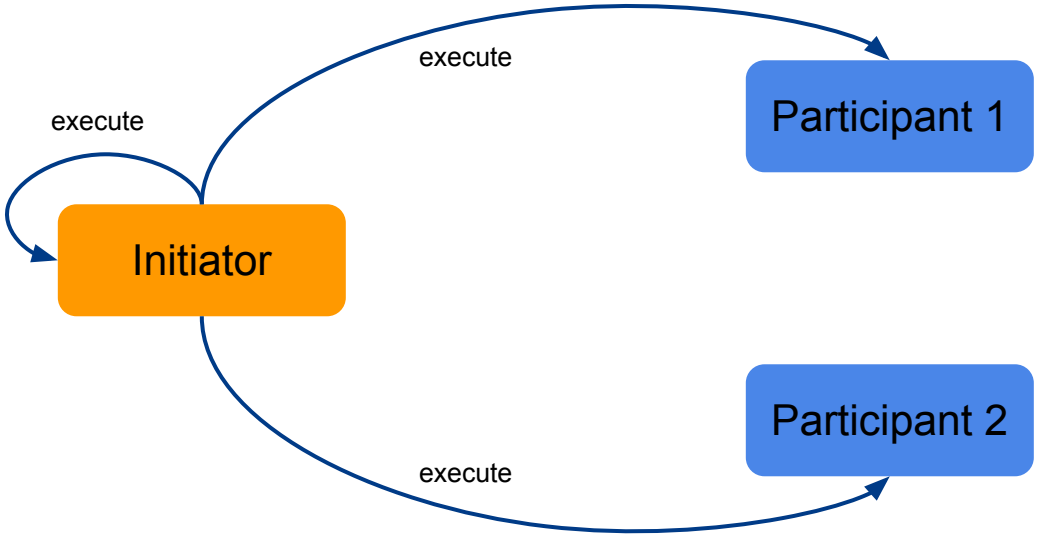
Минимальный протокол



- План содержит список узлов, на котором он должен быть исполнен
- Роли: **initiator** и **participant**

hazelcast

Минимальный протокол



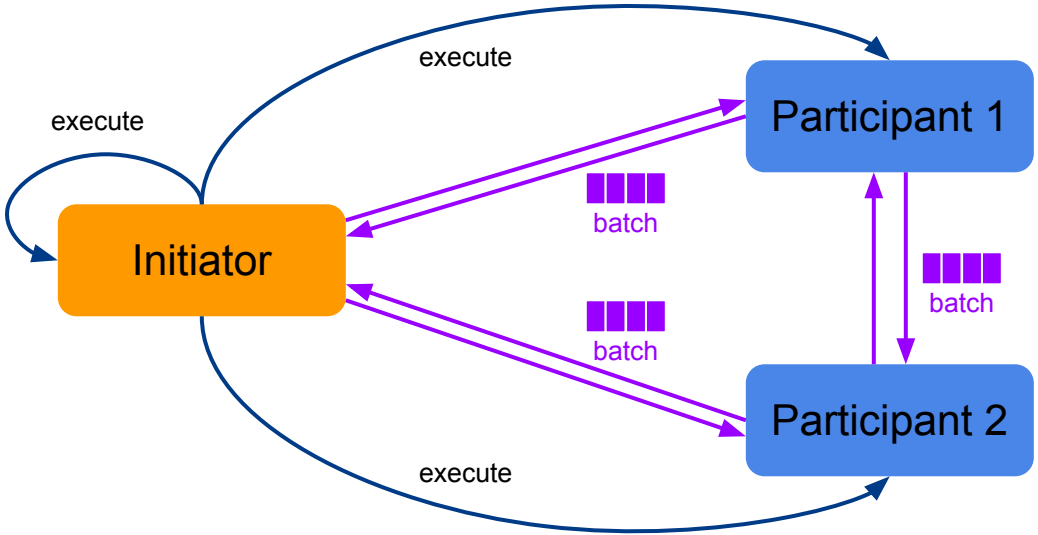
- План содержит список узлов, на котором он должен быть исполнен
- Роли: **initiator** и **participant**

Минимальный протокол:

- **execute** - начать выполнение запроса на узлах-участниках

hazelcast

Минимальный протокол



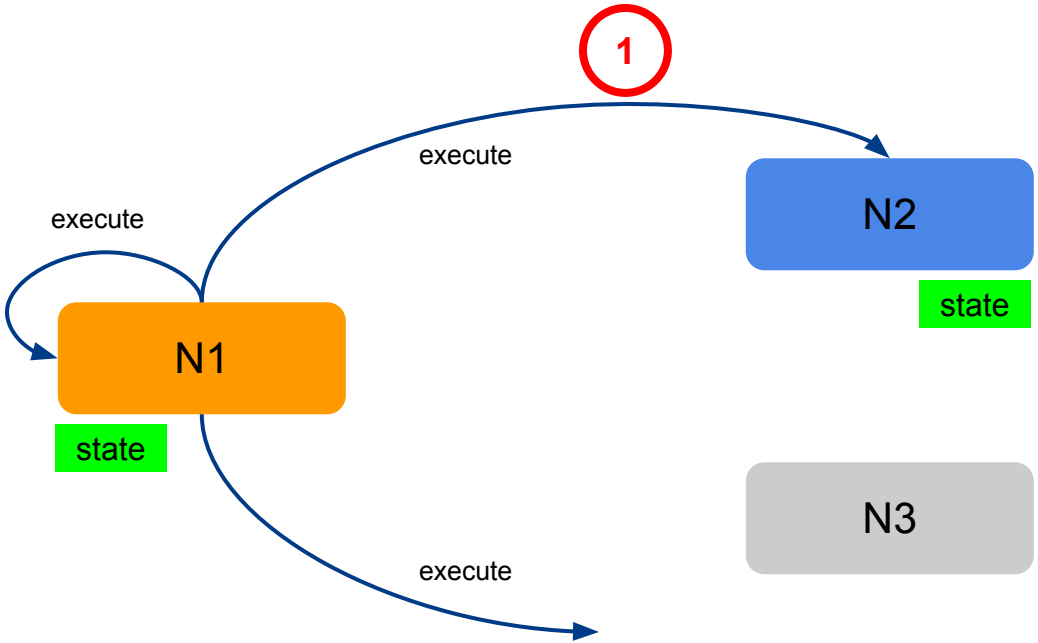
- План содержит список узлов, на котором он должен быть исполнен
- Роли: **initiator** и **participant**

Минимальный протокол:

- **execute** - начать выполнение запроса на узлах-участниках
- **batch** - обмен данными между участниками

hazelcast

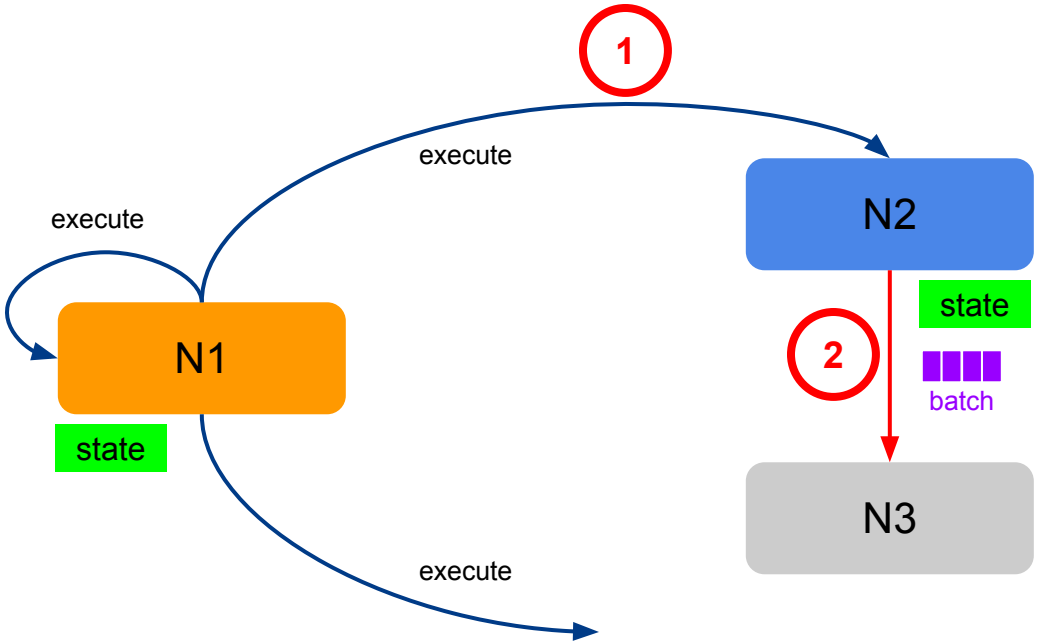
Запуск запроса: гонка



1. **N2** получил `execute`

hazelcast

Запуск запроса: гонка

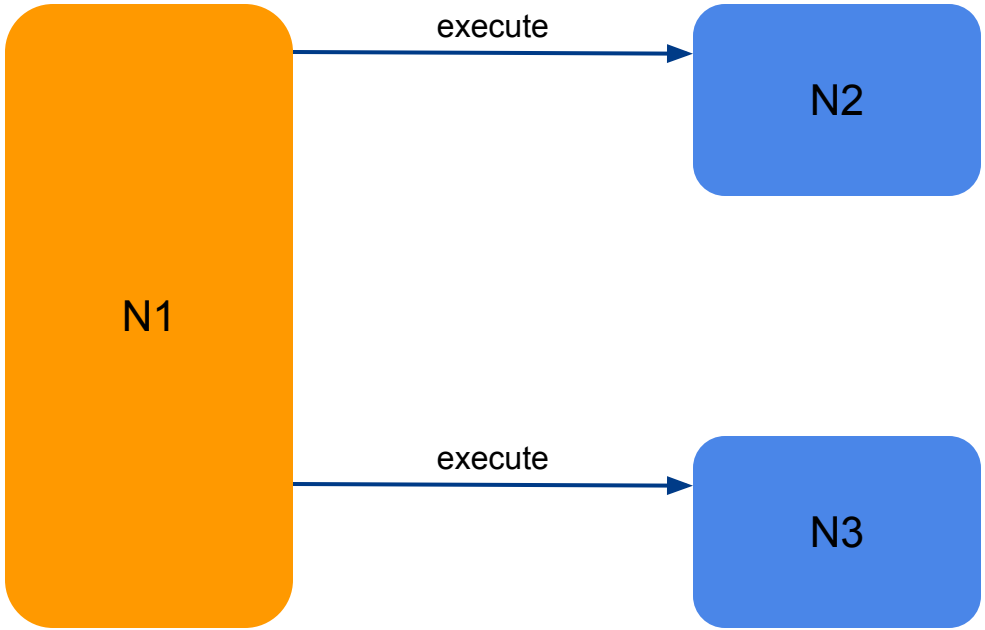


1. **N2** получил *execute*
2. **N2** отправил *batch* **N3**, но **N3** еще не получил *execute*

Что делать с этим батчем?

hazelcast

Двухфазный запуск

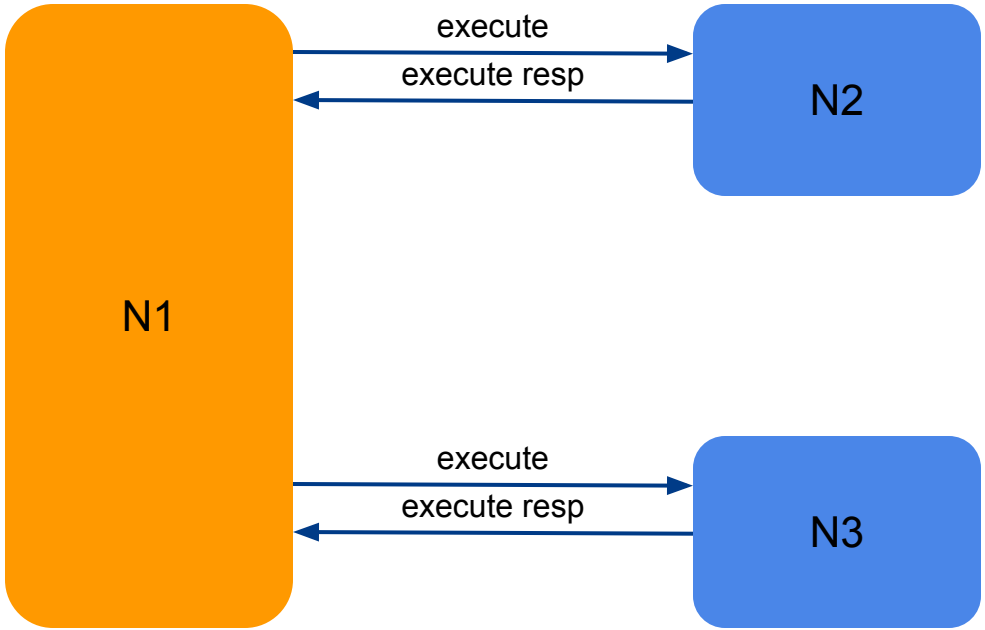


Протокол:

1. Отправить `execute`

hazelcast

Двухфазный запуск

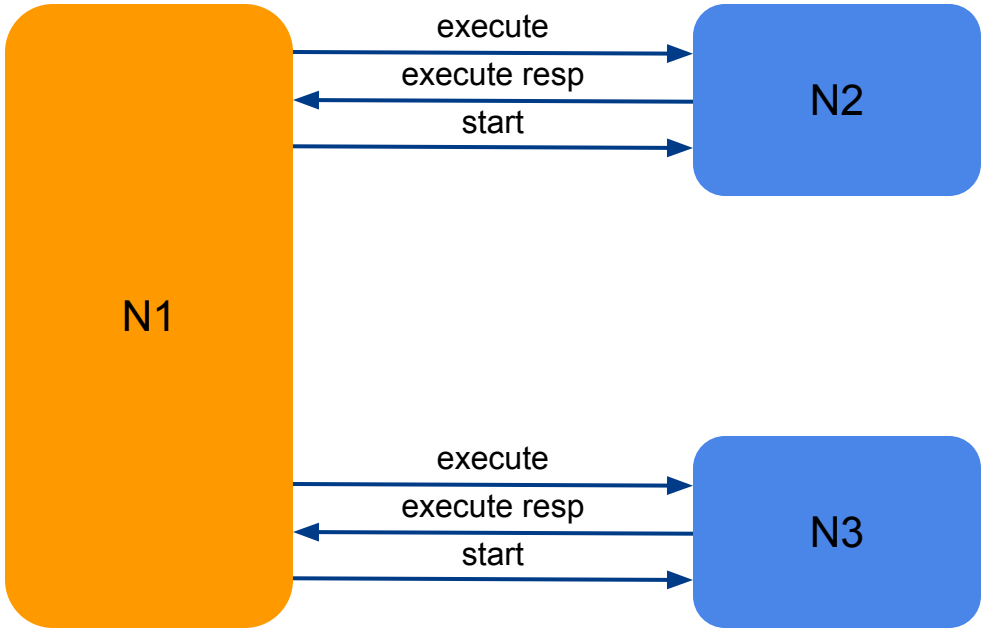


Протокол:

1. Отправить `execute`
2. Дождаться от всех ответа

hazelcast

Двухфазный запуск

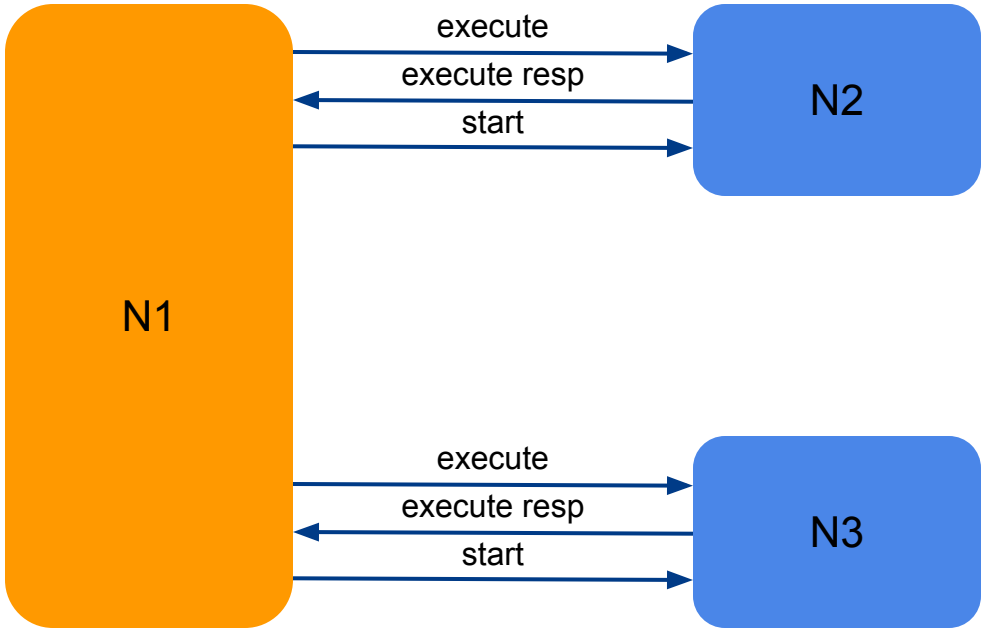


Протокол:

1. Отправить `execute`
2. Дождаться от всех ответа
3. Отправить `start`

hazelcast

Двухфазный запуск



Протокол:

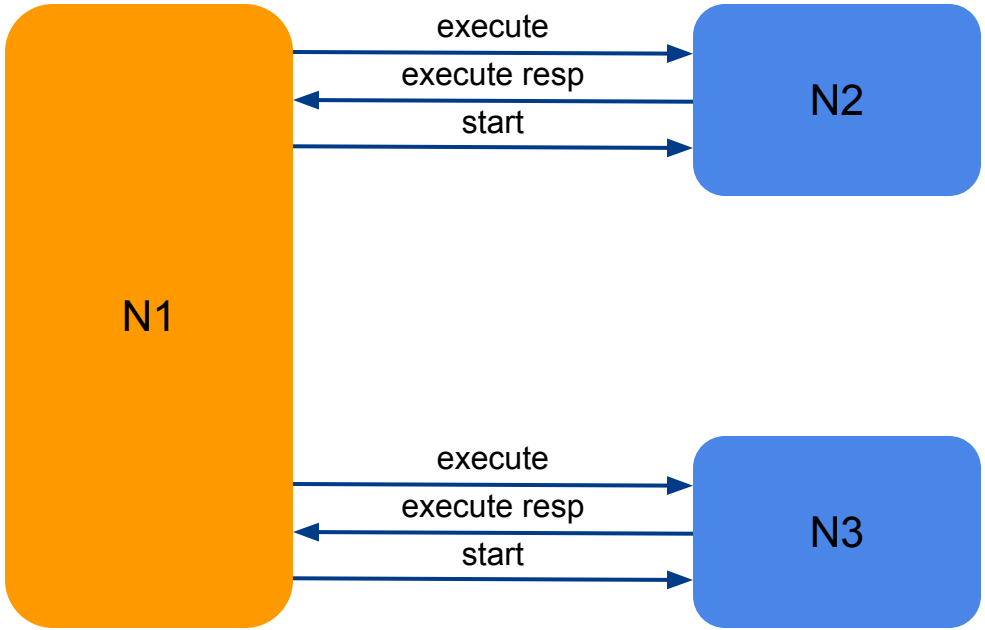
1. Отправить `execute`
2. Дождаться от всех ответа
3. Отправить `start`

Следствие:

- `execute` happens-before `start`

hazelcast

Двухфазный запуск



Протокол:

1. Отправить `execute`
2. Дождаться от всех ответа
3. Отправить `start`

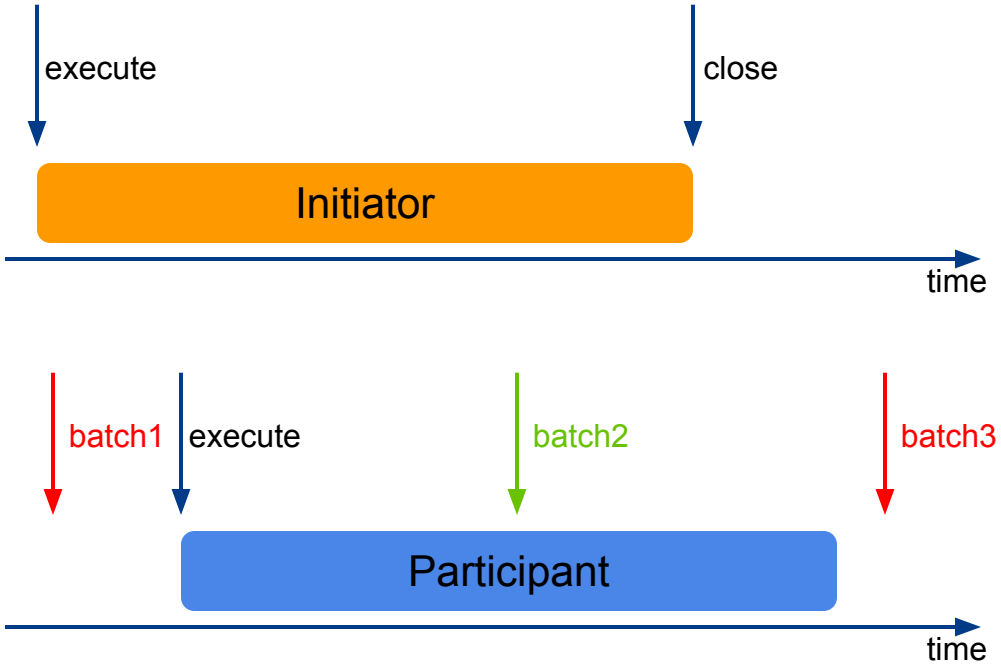
Следствие:

- `execute` happens-before `start`
- Долгий старт (+1 RTT)

НЕ ПОДХОДИТ!

hazelcast

Анализ гонки



`batch1`: Запрос пришел до старта запроса - сохраним его в памяти, в ожидании `execute`?

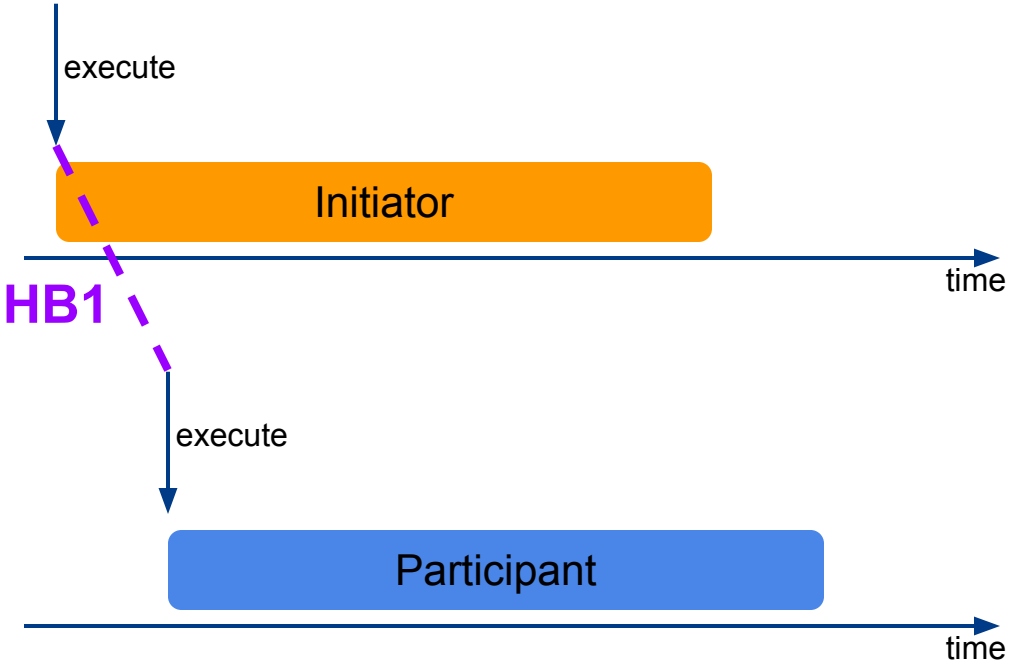
`batch2`: Запрос пришел после старта запроса - ОК!

`batch3`: Запрос пришел после окончания запроса - игнор?

Проблема: `batch1` и `batch3` **неотличимы** друг от друга!

hazelcast

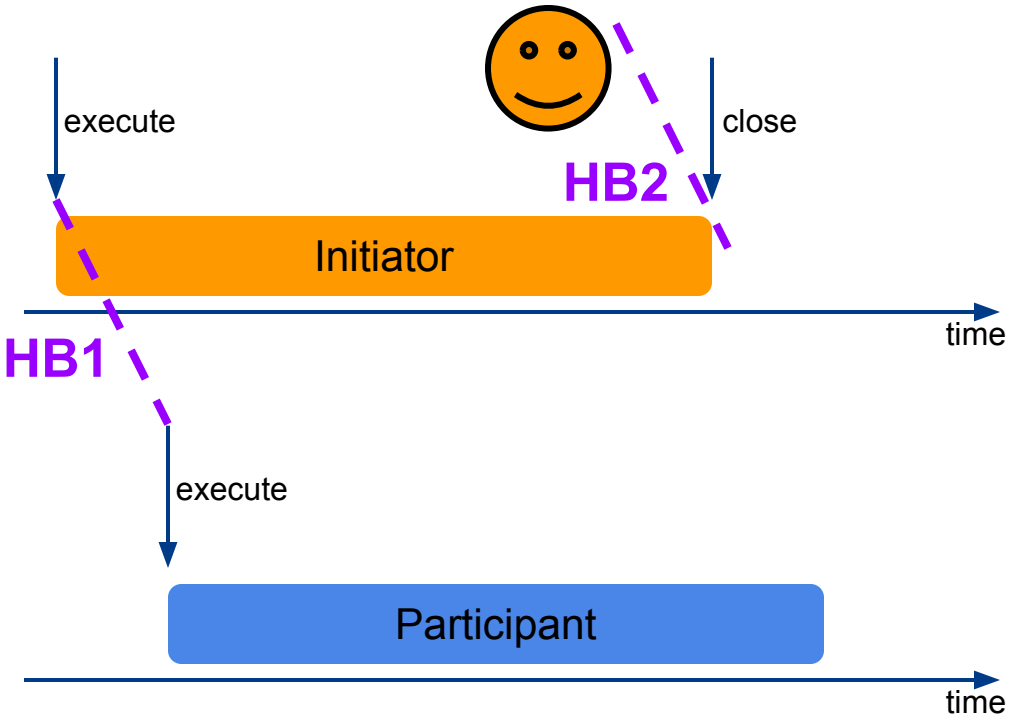
Happens-before



HB1: Запрос на *initiator* стартует раньше, чем на *participant*

hazelcast

Happens-before

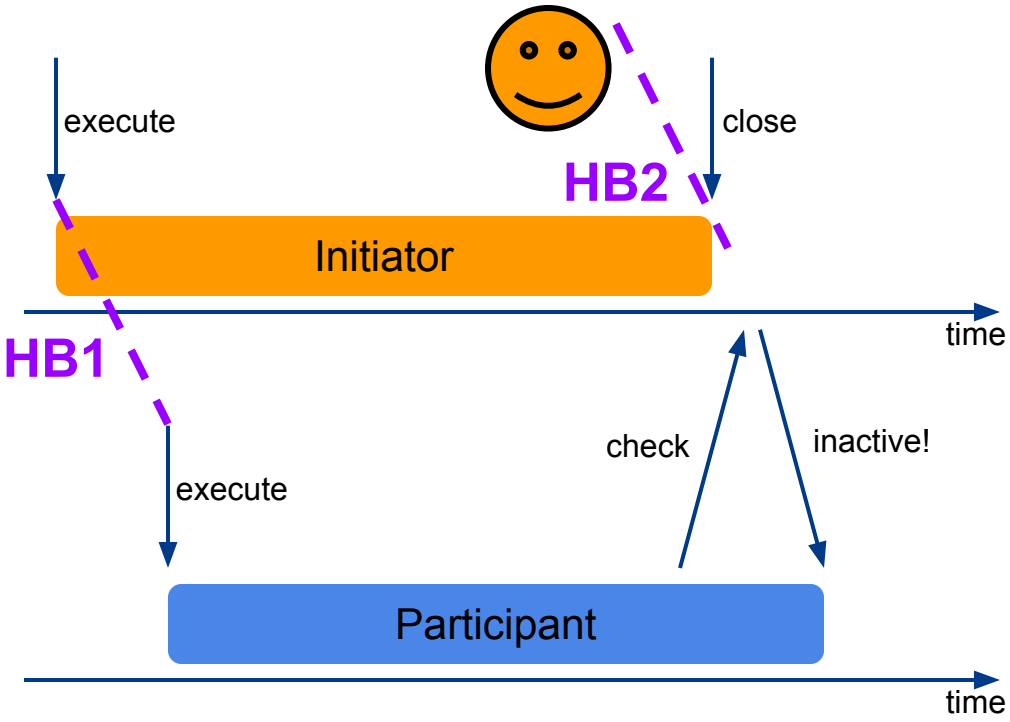


HB1: Запрос на *initiator* стартует раньше, чем на *participant*

HB2: После закрытия запроса на инициаторе, никакое действие не может повлиять на результат запроса

hazelcast

Happens-before



HB1: Запрос на *initiator* стартует раньше, чем на *participant*

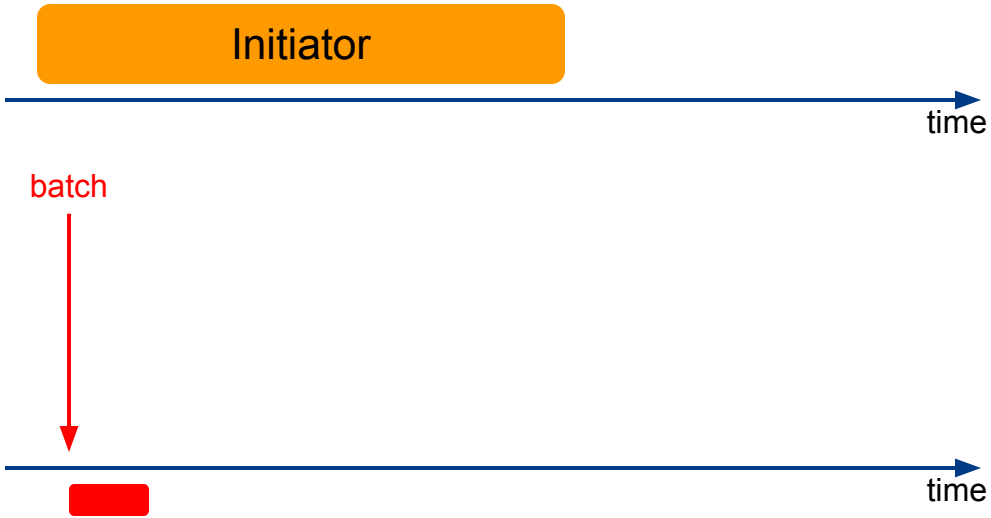
HB2: После закрытия запроса на инициаторе, никакое действие не может повлиять на результат запроса

Благодаря **HB1** и **HB2** *participant* может гарантированно узнать, активен ли еще запрос

hazelcast

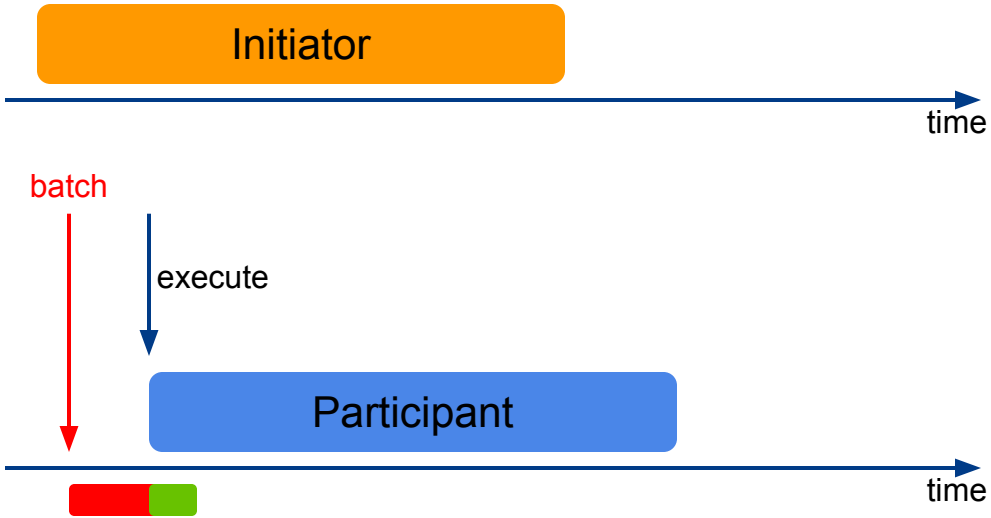
Решение

1. Сохраняем `batch` в памяти



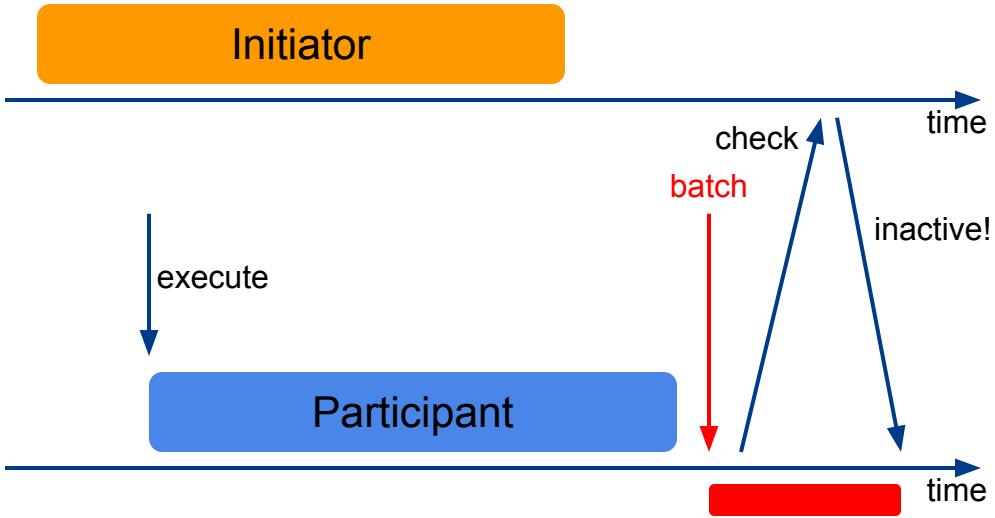
hazelcast

Решение



1. Сохраняем `batch` в памяти
2. Если `execute` пришел, то берем сохраненные `batch` в работу

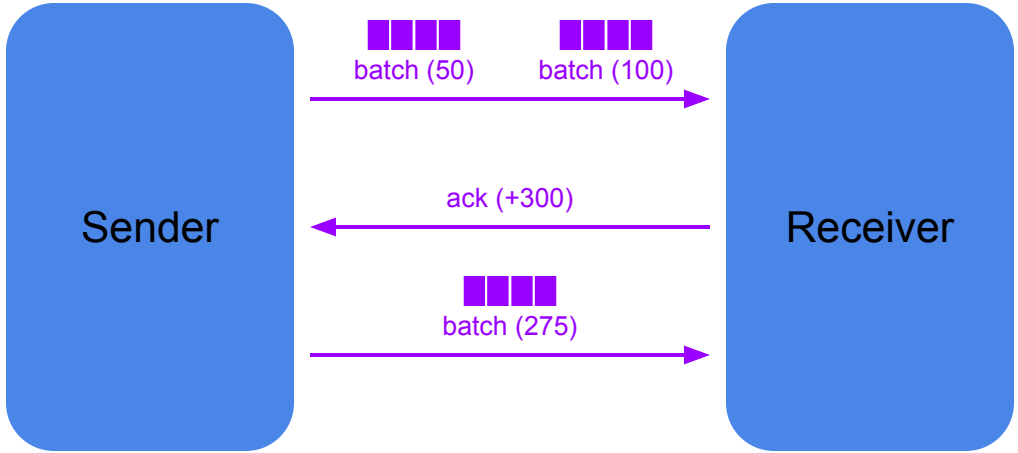
Решение



1. Сохраняем `batch` в памяти
2. Если `execute` пришел, то берем сохраненные `batch` в работу
3. Если `execute` не пришел, инициируем `check`
4. Если `check` показал, что запрос неактивен - удаляем сохраненный `batch`

hazelcast

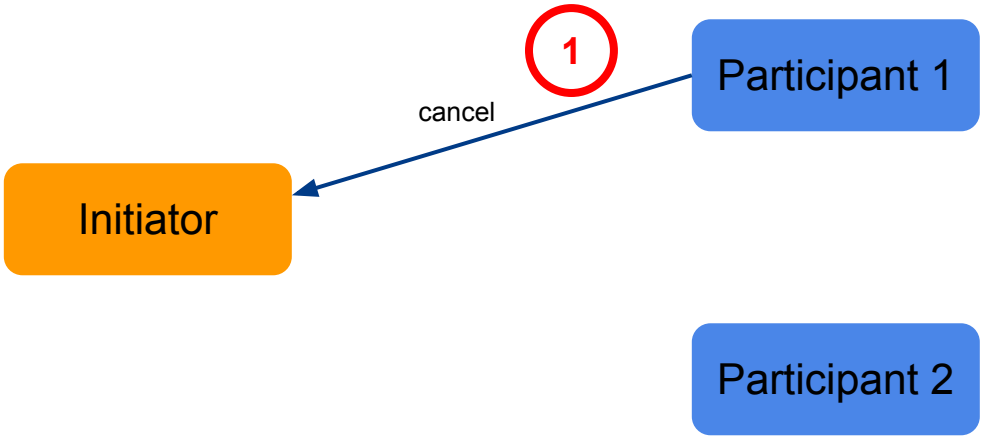
Flow control: как не перегрузить receiver?



- Узлы договариваются о начальном количестве “кредитов”
- **Sender**
  - Уменьшает количество кредитов при отправке очередного пакета
  - Перестает отправлять пакеты при достижении нуля
- **Receiver**
  - Периодически отправляет **ack** на sender, увеличивая количество кредитов

hazelcast

Отмена запроса



Причины

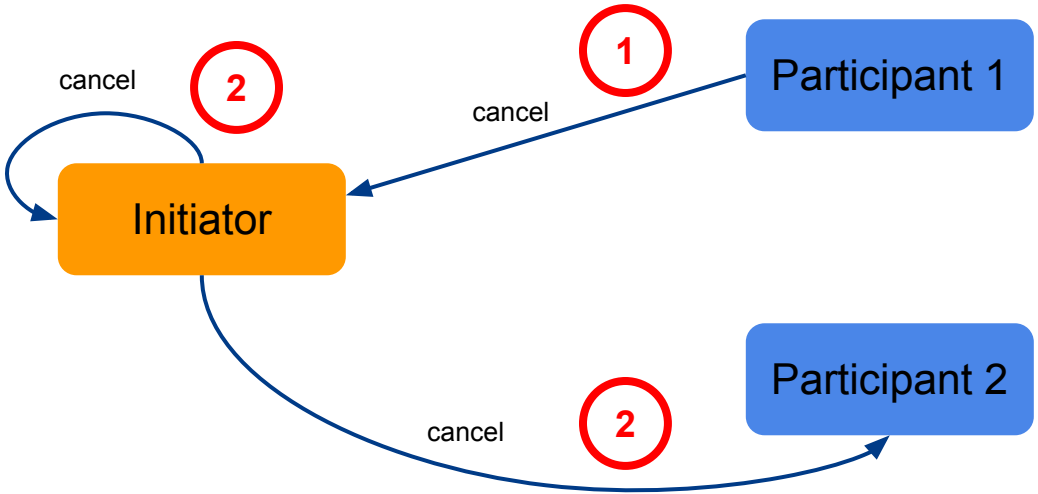
- Запрос пользователя
- Таймаут
- Ошибка

Протокол:

- Какой-то узел инициирует отмену, отправляя `cancel` на **initiator**

hazelcast

Отмена запроса



Причины

- Запрос пользователя
- Таймаут
- Ошибка

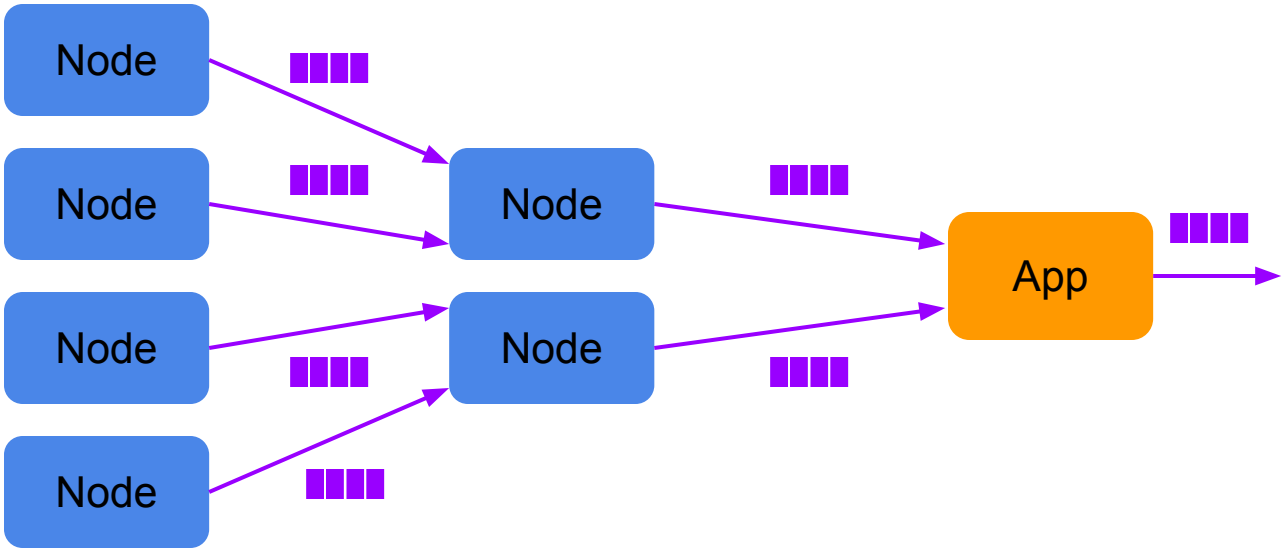
Протокол:

- Какой-то узел инициирует отмену, отправляя `cancel` на **initiator**
- **initiator** рассылает `cancel` остальным участникам

`2*(N-1)` сообщений в худшем случае

hazelcast

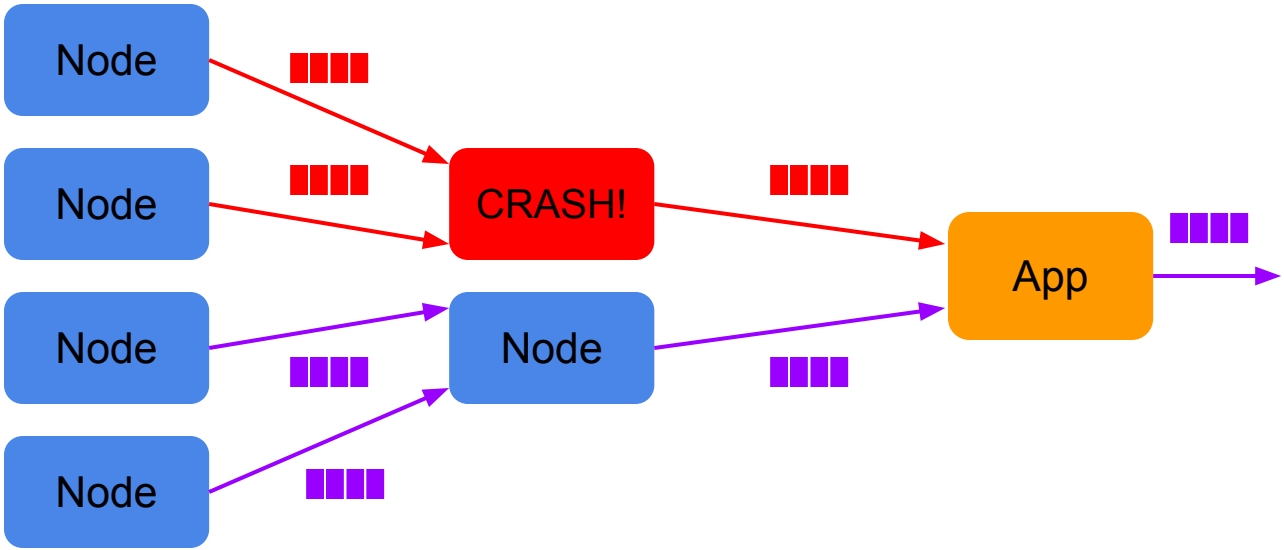
Отказоустойчивость



- Выполнение запроса - это **stateful**-процесс

hazelcast

Отказоустойчивость



- Выполнение запроса - это **stateful**-процесс
- Для продолжения работы при произвольном отказе необходимо отслеживать, какие данные куда ушли!
  - Иначе результат может быть неполным или содержать дубликаты

hazelcast

Отказоустойчивость

Что делать, если во время выполнения запроса произошло изменение конфигурации кластера (упал узел, переехал partition, …)?

- Прервать выполнение запроса
  - Легко реализовать
  - Неудобно для пользователей
- Попытаться продолжить выполнение
  - Тяжело: как найти точку, с которой продолжить выполнение?
  - Комфортнее для пользователей
- Наш ответ: **падаем, но будем улучшать**

Итого

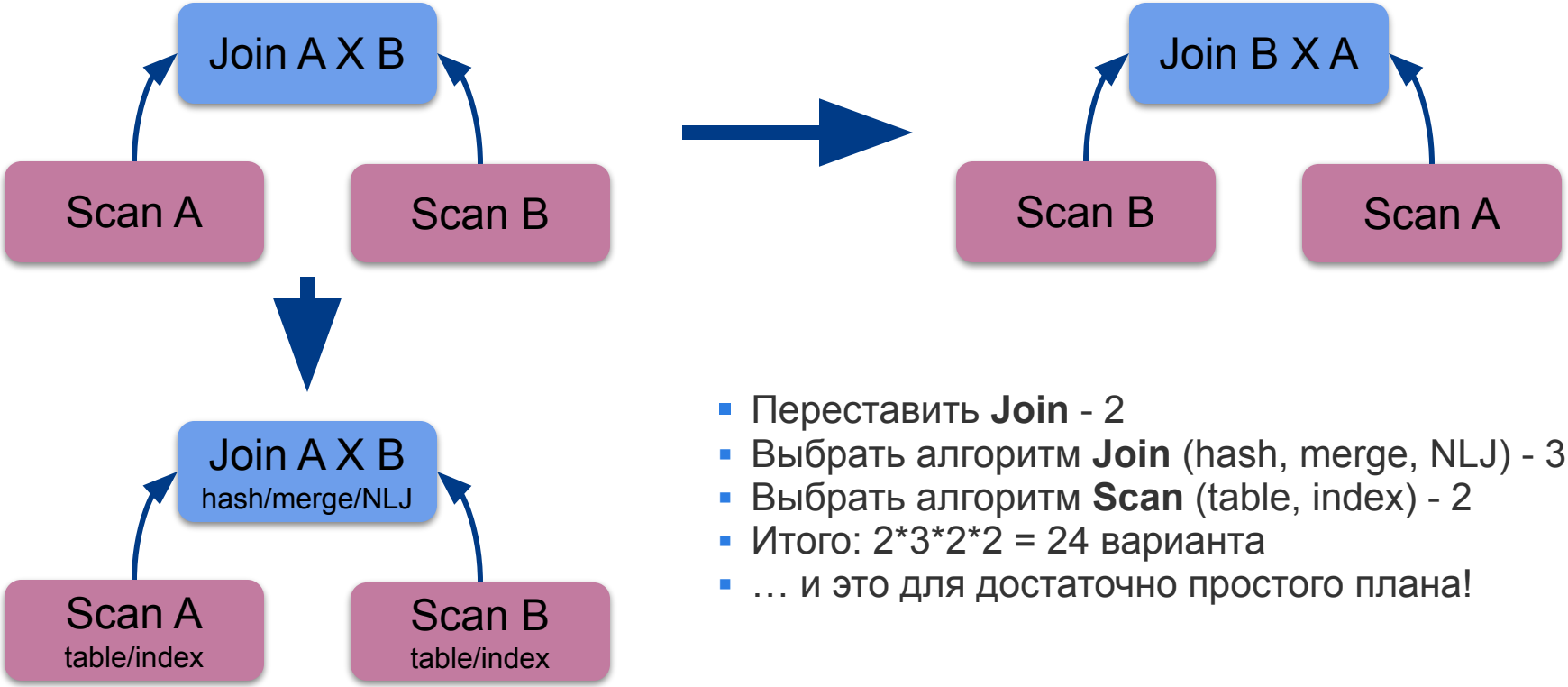
- Философия: минимум ожидания
  - Однофазный старт
  - Стриминговый обмен данными между узлами с flow control
- Просим пользователя перезапустить запрос в случае изменения конфигурации кластера
  - Будем постепенно улучшать

Выводы

- Создание распределенного SQL-движка - это **тяжело** и **долго**
  - Готовьтесь много исследовать
  - Мы уже потратили полтора года, и конца пока не видно
- Apache Calcite может вас хорошо ускорить, но его использование сопряжено с определенной **болью** (и это нормально)
- Тем не менее, мы считаем эти усилия оправданными, так как SQL значительно расширяет применимость Hazelcast IMDG

Thank You

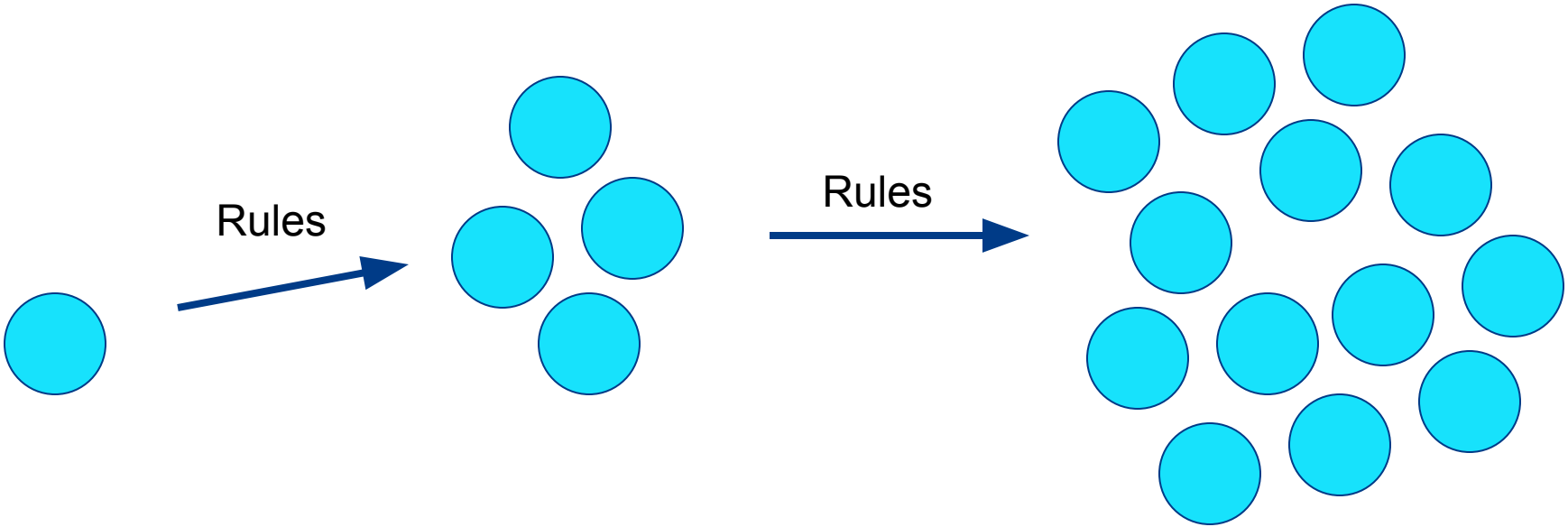
Пространство поиска



- Переставить **Join** - 2
- Выбрать алгоритм **Join** (hash, merge, NLJ) - 3
- Выбрать алгоритм **Scan** (table, index) - 2
- Итого: 2*3*2*2 = 24 варианта
- … и это для достаточно простого плана!

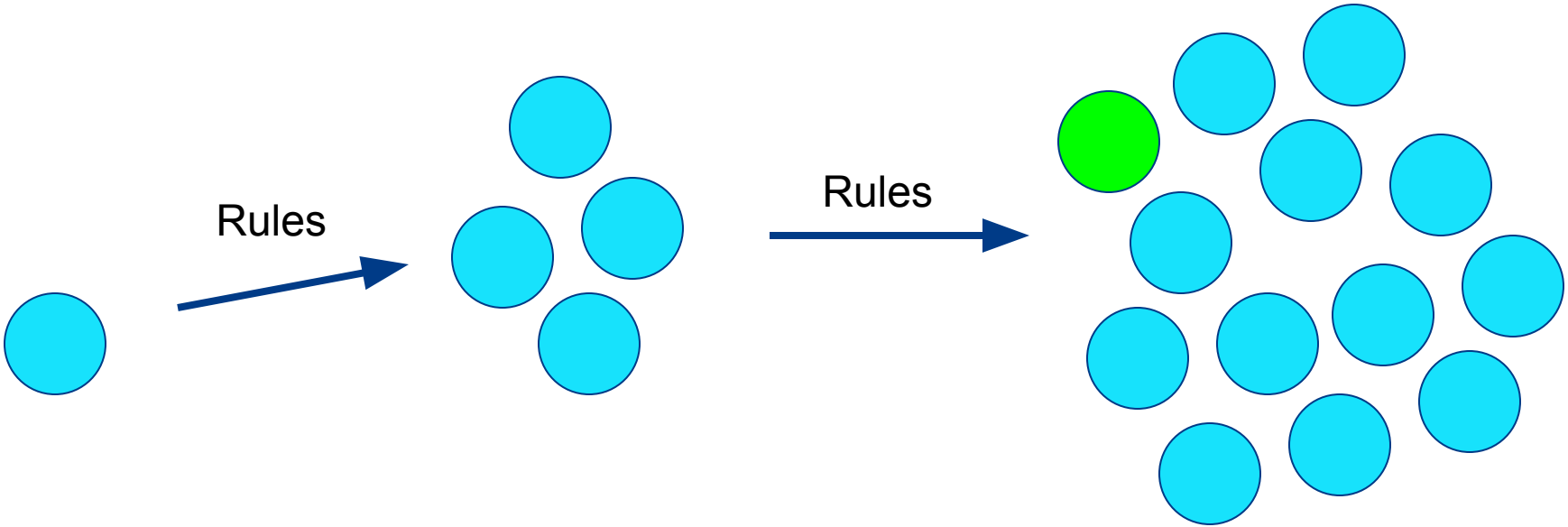
hazelcast

Фазы оптимизации



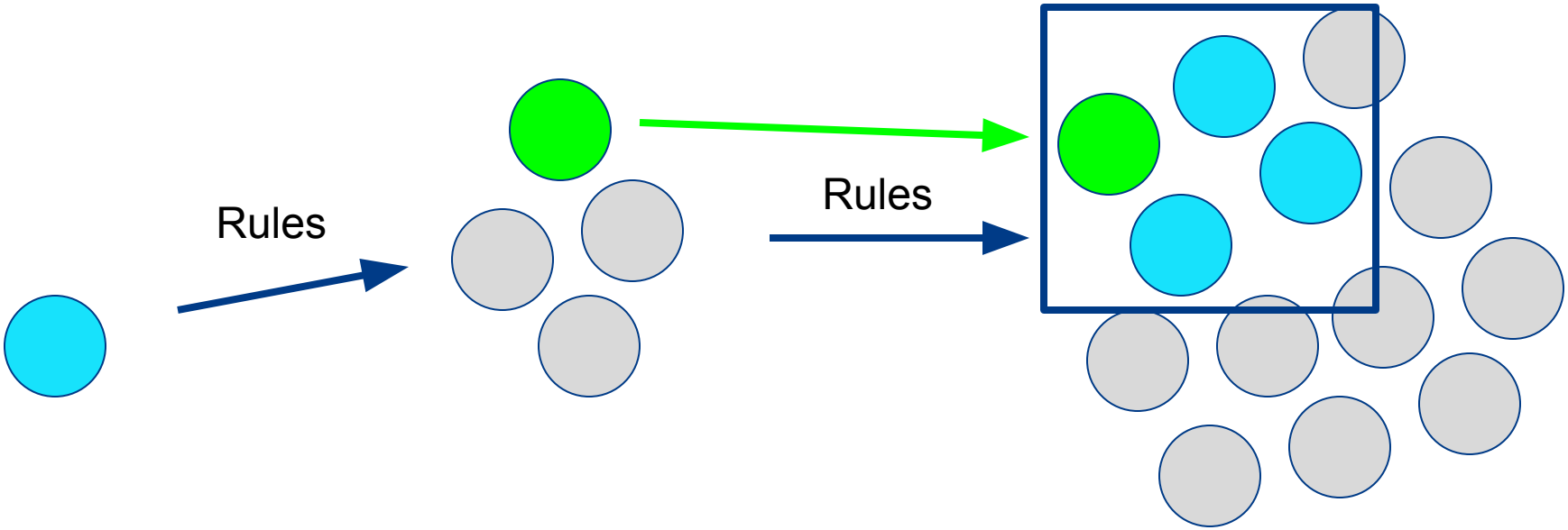
hazelcast

Фазы оптимизации



hazelcast

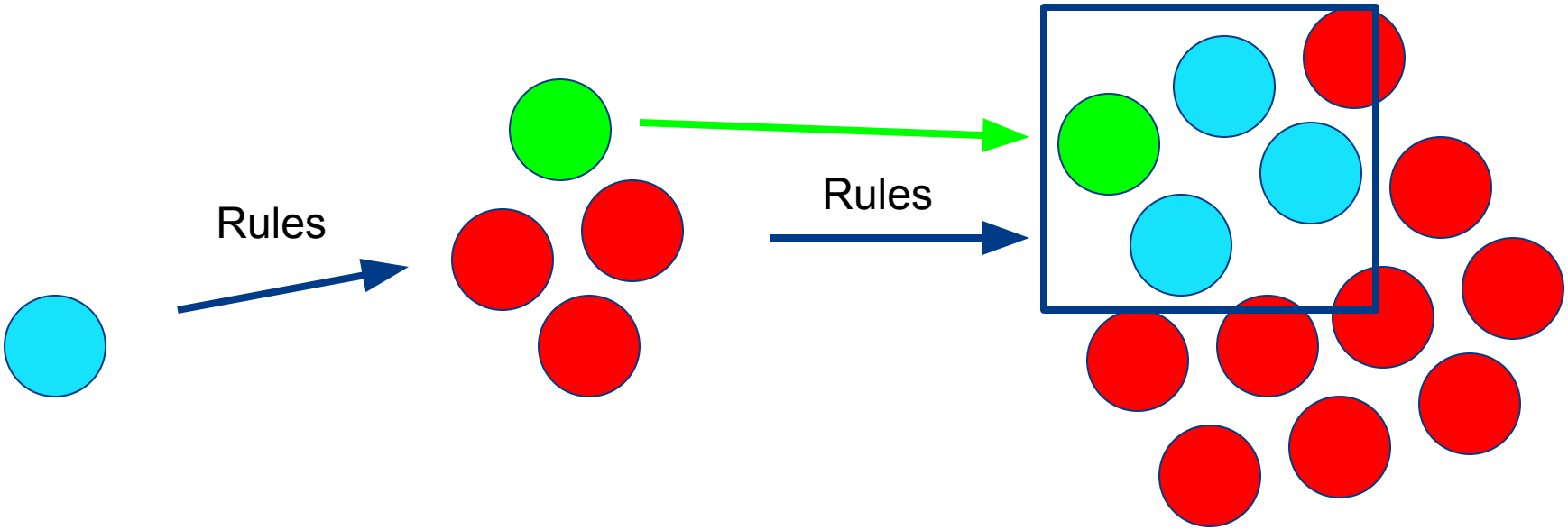
Фазы оптимизации



Иногда, в середине оптимизации можно сказать, что какие-то промежуточные планы (почти) точно не создадут лучший план

hazelcast

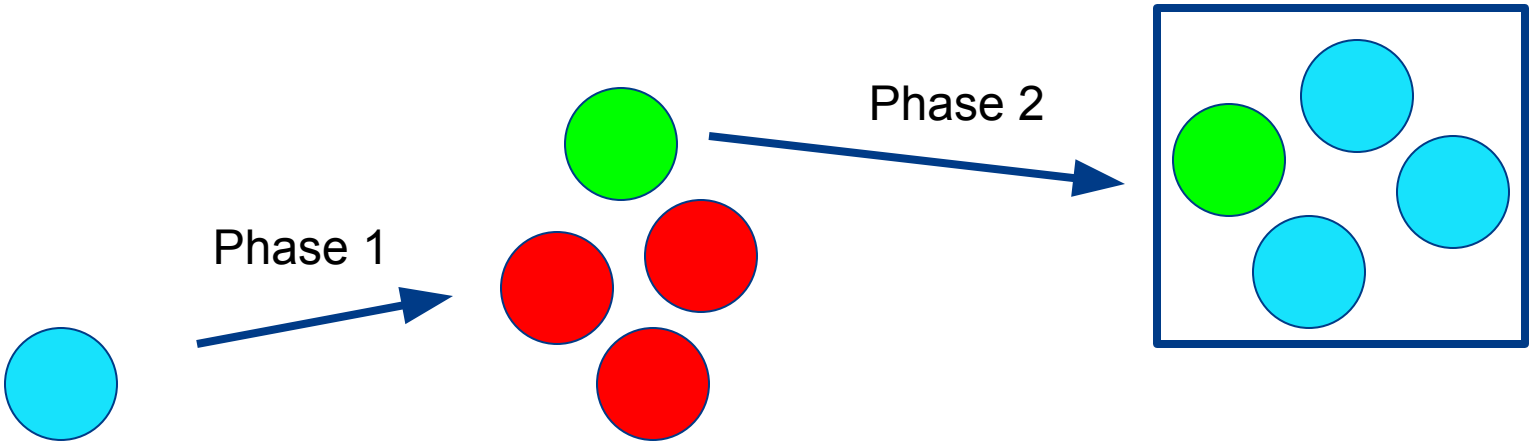
Фазы оптимизации



Иногда, в середине оптимизации можно сказать, что какие-то промежуточные планы (почти) точно не создадут лучший план

hazelcast

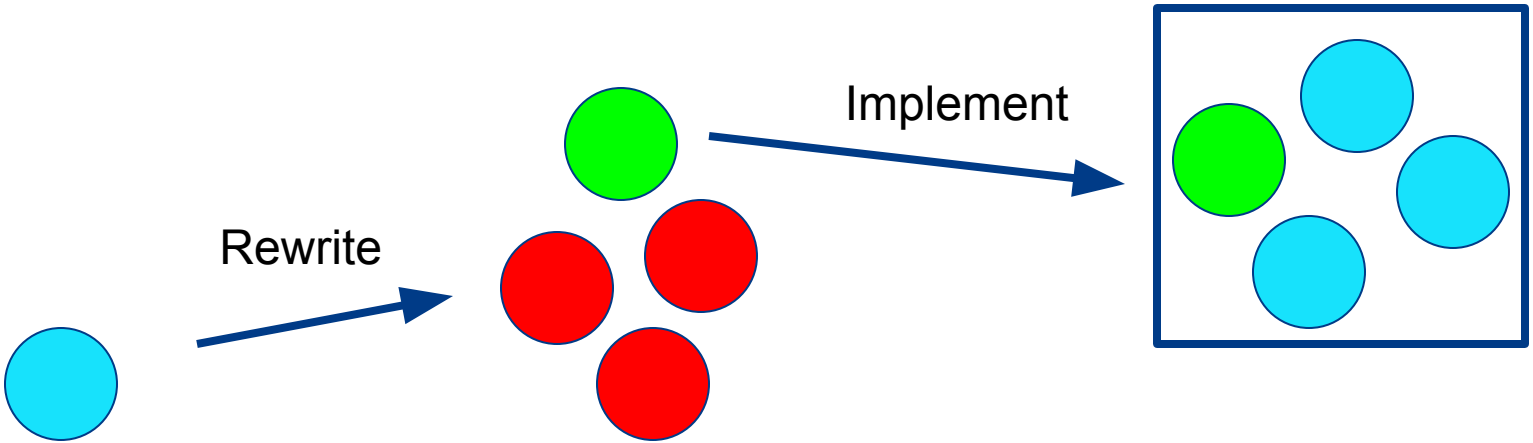
Фазы оптимизации



Можно разбить оптимизацию на фазы. Для каждой из них:

- Генерируем ограниченное число альтернатив
- Выбираем один наилучший план
- Переходим на следующую фазу

Фазы оптимизации

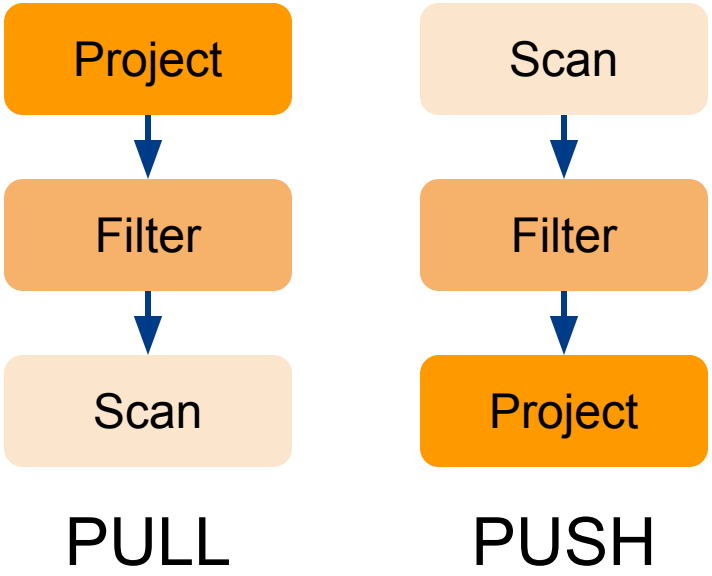


Типичные фазы:

- Rewrite: применить универсально “хорошие” правила
- Implement (aka): применить все остальное

Volcano: можно ли лучше?

Efficiently Compiling Efficient Query Plans for Modern Hardware (2011)

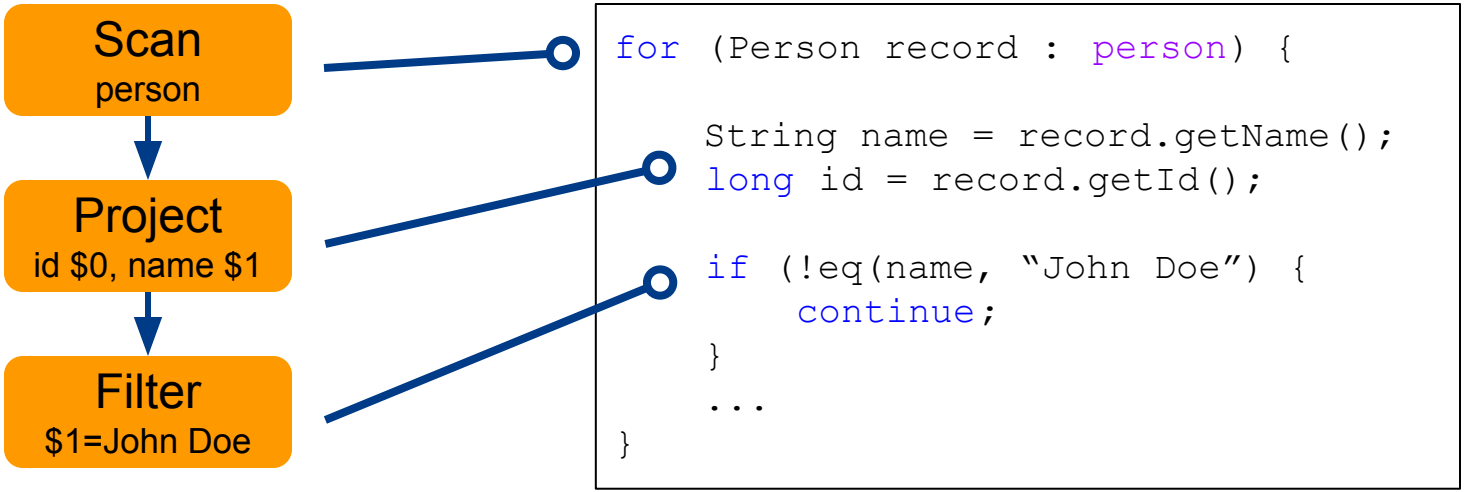


- Меняем направление данных: **push** vs **pull**
- Компилируем!

hazelcast

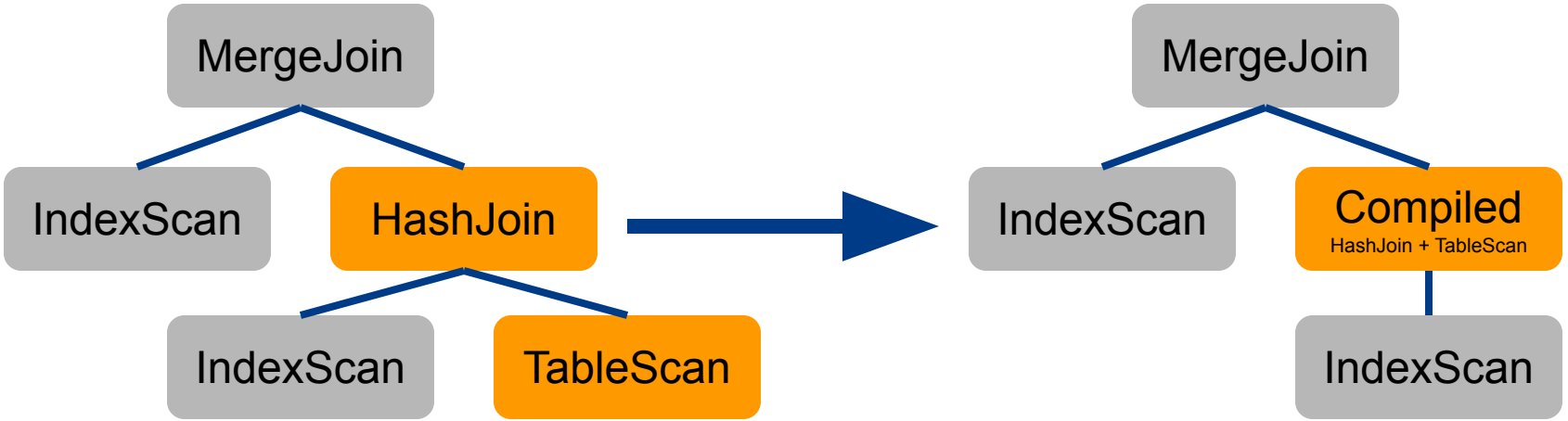
Push-based итерация с компиляцией

```
SELECT id, name
FROM person
WHERE name = 'John Doe'
```



hazelcast

Наш прототип компилятора



- Операторы могут быть скомпилированы как независимо друг от друга, так и группой
- Генерируем исходный код, компилируем с помощью **Janino**
- Встраиваем новый оператор во фрагмент

hazelcast