



**60029**

**Data Processing Systems  
Imperial College London**

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Logistics . . . . .	4
1.2	Data Management Systems . . . . .	4
1.3	Data Intensive Applications . . . . .	5
1.4	Data Management Systems . . . . .	6
1.4.1	Non-Functional Requirements . . . . .	6
1.4.2	Logical/Physical Data Model Separation . . . . .	7
1.4.3	Transactional Concurrency . . . . .	7
1.4.4	Read Phenomena . . . . .	8
1.4.5	Isolation levels . . . . .	8
1.4.6	Declarative Data Analysis . . . . .	8
<b>2</b>	<b>Relational Algebra</b>	<b>9</b>
2.1	Relational Structures . . . . .	9
2.1.1	Preliminaries . . . . .	9
2.1.2	Nomenclatures . . . . .	11
2.2	Implementing Relational Algebra in C++ . . . . .	11
2.2.1	Relation . . . . .	11
2.2.2	Project . . . . .	12
2.2.3	Select . . . . .	12
2.2.4	Cross Product / Cartesian . . . . .	13
2.2.5	Union . . . . .	13
2.2.6	Difference . . . . .	13
2.2.7	Group Aggregation . . . . .	14
2.2.8	Top-N . . . . .	14
<b>3</b>	<b>Storage</b>	<b>16</b>
3.1	Database Management System Kernel . . . . .	16
3.2	Storage . . . . .	16
3.2.1	Storage Manager . . . . .	16
3.2.2	Catalog . . . . .	18
3.2.3	Disk Storage . . . . .	18
3.3	Example Sketch Implementation . . . . .	20
<b>4</b>	<b>Algorithms and Indices</b>	<b>21</b>
4.1	Sorting Algorithms . . . . .	21
4.1.1	Quicksort . . . . .	21
4.1.2	Merge Sort . . . . .	21
4.1.3	Tim Sort . . . . .	21
4.1.4	Radix Sort . . . . .	21
4.1.5	Top-N with Heaps . . . . .	21
4.2	Joins . . . . .	21
4.2.1	Database Normalisation . . . . .	21
4.2.2	Join Types . . . . .	21
4.2.3	Join Implementations . . . . .	23
4.2.4	Nested Loop Join . . . . .	24
4.2.5	Sort Merge Join . . . . .	24
4.2.6	Hash Join . . . . .	25
4.3	Hash Tables . . . . .	26

4.3.1	Probing Hashmap . . . . .	26
4.3.2	Basic Hash Table Implementation . . . . .	29
4.3.3	Partitioning . . . . .	34
4.3.4	Indexing . . . . .	34
4.3.5	Hash Indexes . . . . .	35
4.3.6	Bitmap Indexing . . . . .	35
4.3.7	B-Trees . . . . .	37
4.3.8	B+ Trees . . . . .	38
4.3.9	Foreign Key Indices . . . . .	38
<b>5</b>	<b>Velox</b>	<b>39</b>
5.1	Motivation . . . . .	39
5.2	Overview . . . . .	39
5.2.1	Structure . . . . .	39
5.3	Use Cases . . . . .	39
5.4	Library Components . . . . .	39
5.4.1	Data Types . . . . .	39
<b>6</b>	<b>Processing Models</b>	<b>40</b>
6.1	Motivation . . . . .	40
6.2	Volcano Processing . . . . .	41
6.2.1	Operators . . . . .	41
6.2.2	Pipelining . . . . .	50
6.2.3	Operations Calculations . . . . .	52
6.3	Bulk Processing . . . . .	52
6.3.1	By-Reference Bulk Processing . . . . .	54
6.3.2	Decomposed Bulk Processing . . . . .	55
<b>7</b>	<b>Optimisation</b>	<b>56</b>
7.1	Motivation . . . . .	56
7.1.1	Query Optimisers vs Optimising Compilers . . . . .	56
7.1.2	Query Equivalence . . . . .	57
7.2	Peephole Transformations . . . . .	57
7.2.1	Avoiding Cycles . . . . .	58
7.2.2	Branches . . . . .	58
7.3	Classifying Optimisation . . . . .	59
7.4	Logical Optimisation . . . . .	59
7.4.1	Rule Based Logical Optimisation . . . . .	60
7.4.2	Cost Based Logical Optimisation . . . . .	62
7.5	Physical Optimisation . . . . .	63
7.5.1	Rule Based Physical Optimisation . . . . .	63
7.5.2	Cost Based Physical Optimisation . . . . .	64
7.6	SparkSQL . . . . .	64
<b>8</b>	<b>Transactions</b>	<b>65</b>
8.1	SQL Transaction . . . . .	65
8.1.1	ACID Properties . . . . .	65
8.2	Histories . . . . .	66
8.3	Anomalies . . . . .	67
8.4	Isolation Levels . . . . .	68
8.5	Concurrency Schemes . . . . .	69
8.5.1	Serial Execution . . . . .	69
8.5.2	Two-Phase Locking (2PL) . . . . .	70
8.5.3	Timestamp Ordering . . . . .	71
8.5.4	Optimistic Concurrency Control (OCC) . . . . .	71
8.5.5	Multi-Version Concurrency Control (MVCC) . . . . .	71

<b>9 Streams</b>	<b>72</b>
9.1 Motivation . . . . .	72
9.2 Push Operators . . . . .	72
9.2.1 Naive Implementation . . . . .	72
9.2.2 PushBack . . . . .	74
9.3 Time . . . . .	74
9.3.1 In-Order Processing . . . . .	75
9.3.2 Windows . . . . .	76
9.3.3 Aggregate Implementations . . . . .	77
9.3.4 Two Stacks Algorithm . . . . .	79
9.4 Stream Joins . . . . .	81
9.4.1 Handshake Join . . . . .	81
9.4.2 Symmetric Hash-Joins . . . . .	81
9.4.3 Bloom Filters . . . . .	82
<b>10 Advanced Topics</b>	<b>83</b>
<b>11 Credit</b>	<b>84</b>

# Chapter 1

## Introduction

### 1.1 Logistics

#### A note on types...

#### Extra Fun! 1.1.1

In real data processing systems (and in particular databases), types of data are not known at runtime (i.e do not know the types of columns, tables until they are created, amended, and operated on at runtime).

For simplicity in many code examples the types of data will be encoded through templates, and types at compile time (change a table or query requires the example to be recompiled).

### 1.2 Data Management Systems

#### Database

#### Definition 1.2.1

A large collection of organized data.

- Can apply to any structured collection of data (e.g a relational table, data structures such as vectors & sets, graphs etc.)

#### System

#### Definition 1.2.2

A collection of components interacting to achieve a greater goal.

- Usually applicable to many domains (e.g a database, operating system, webserver). The goal is domain-agnostic
- Designed to be flexible at runtime (deal with other interacting systems, real conditions) (e.g OS with user input, database with varying query volume and type)
- Operating conditions are unknown at development time (Database does not know schema prior, OS does not know number of users prior, Tensorflow does not know matrix dimensionality prior)

Large & complex systems are typically developed over years by multiple teams.

#### Data Management System Definition 1.2.3

A system built to control the entire lifecycle of some data.

- Creation, modification, inspection and deletion of data
- Classic examples include *Database Management Systems*

#### Data Processing System

#### Definition 1.2.4

A system for processing data.

- Support part of the data lifecycle
- A strict superset of Data Management Systems (all data management systems are data processing systems)

For example a tool as small as `grep` could be considered a data processing system.

Building data management systems is hard!

- Often must fetch data continuously from multiple sources
- Needs to be highly reliable (availability/low downtime & data retention)
- Needs to be efficient (specification may contain performance requirements)

<b>Storage</b>	Needs to be persistent (but also needs to be fast)
<b>Data Ingestions</b>	Needs to allow for easy import of data (e.g by providing a csv, another database's url)
<b>Concurrency</b>	To exploit parallelism in hardware (e.g multithreaded, distributed over several machines)
<b>Data Analysis</b>	For inspection (typically the reason to hold data in first place)
<b>Standardized Programming Model</b>	Features are not implemented in an ad-hoc way but through common abstractions, users and developers do not need to radically change how they approach a new feature.
<b>User Defined Functions</b>	
<b>Access Control</b>	Not all data is shared between all users.
<b>Self-Optimization</b>	Monitors its own workloads in an attempt to optimise (e.g keeping frequently accessed data in memory)

## 1.3 Data Intensive Applications

Data Intensive Application	Definition 1.3.1
An application that acquires, stores and processes a significant amount of information. Core functionality of the application is based on data.	

There are several common patterns for data-intensive applications:

### Online Transaction Processing (OTP)

- High volume of small updates to a persistent database
- ACID is important

Goal: Throughput

### Online Analytical Processing (OLAP)

- Running a single data analysis task.
- A mixture of
- Queries are ad-hoc

Goal: Latency

### Reporting

- Running a set of data analysis tasks
- Fixed time budget
- Queries known in advance

Goal: Resource Efficiency

Daily Struggle	Example Question 1.3.1
Provide some examples of <i>Reporting</i> pattern being used in industry.	<ul style="list-style-type: none"><li>• A supermarket getting the day's sales, and stock-take.</li><li>• A trading firm computing their position and logging the day's trades at market-close and informing regulators, clearing, risk department.</li><li>• A company's payroll system running weekly using week long timesheets.</li></ul>

## Hybrid Transactional / Analytical Processing (HTAP)

- Small updates interwoven with larger analytics
- Need to be optimal for combination of small and large task sizes

### HTAP

### Extra Fun! 1.3.1

HTAP is a relatively new pattern used to solve the need for separate systems to work on OTP and OLAP workloads (which introduced complexity and cost as data is frequently copied between the two systems). Read more here.

*Data-Intensive Applications* can be differentiated from *Data Management Systems* (though there is ample ambiguity):

- Applications are domain-specific, and hence contain domain-specific optimisations that prevent fully general-purpose usage
- Data Management Systems are required to be highly generalised
- The cost of application specific data management (e.g developer time) outweighs any benefits for the majority of cases

### Model View Controller (MVC)

### Definition 1.3.2

A common design pattern separating software into components for user interaction (view), action (controller) and storing state (model) which interact.

A typical *data intensive application* has the following architecture:



### Big Business

### Extra Fun! 1.3.2

The enterprise data management systems market has been valued at \$82.25 billion (2021) with annual growth exceeding 10% (grand view research).

## 1.4 Data Management Systems

### 1.4.1 Non-Functional Requirements

<b>Efficiency</b>	Ideally should be as fast as a bespoke, hand-written solution.
<b>Resilience</b>	Must be able to recover from failures (software crashes, power failure, hardware failure)
<b>Robustness</b>	Predictable performance (semantically small change in query $\Rightarrow$ similarly small change in performance)
<b>Scalability</b>	Can scale performance with available resources.
<b>Concurrency</b>	Can serve multiple clients concurrently with a clear model for how concurrency will affect results.

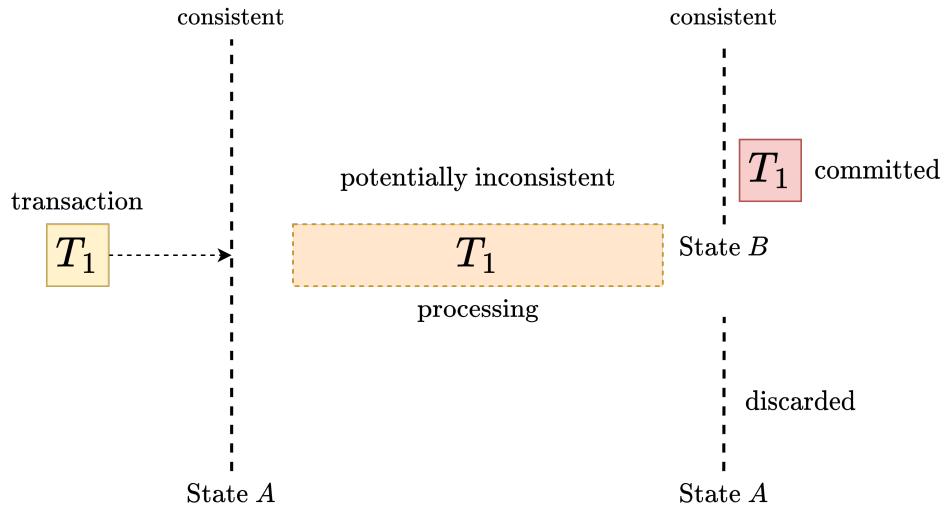
## 1.4.2 Logical/Physical Data Model Separation



## 1.4.3 Transactional Concurrency

Actions to be performed on a data management system can be wrapped up as a *transaction* to be received, processed and committed.

ACID		Definition 1.4.1
A set of useful properties for database management systems.		
<b>Atomic</b>	A transaction either runs entirely (and is committed) or has no effect. (All or nothing)	
<b>Consistent</b>	A transaction can only bring the database from one valid (for some invariants) state to another. Note that there may be inconsistency between.	
<b>Isolated</b>	Many transactions run concurrently, however each leaves the database in some state equivalent to running the transactions in some sequential order. (Run as if alone on the system).	
<b>Durable</b>	Once a transaction is committed, it is persistent (even in case of failure - e.g power failure).	



"Isolated" is the most flexible ACID property, several *isolation levels* describe how concurrent transactions interact. The more isolation is enforced, the more locking is required which can affect performance (contention & blocking).

Concurrency Controls		Extra Fun! 1.4.1
In order to support efficient concurrent access & mutation of data without race conditions concurrency control is used:		
Lock Based	Each object (e.g record, table) contains a lock (read-write) used for synchronisation of access. The most common technique is <i>two-phase locking</i> .	
Multiversion	Each object and transaction is timestamped, by maintaining multiple timestamped versions of an object a transaction can effectively operate on a snapshot of the database at its own timestamp.	

#### 1.4.4 Read Phenomena

Dirty Read / Uncommitted Dependency	Definition 1.4.2
A transaction reads a record updated by a transaction that has not yet committed.	
• The uncommitted transaction may fail or be rolled back rendering the dirty-read data invalid.	
Non-Repeatable Read	Definition 1.4.3
When a transaction reads a record twice with different results (another committed transaction updated the row between the reads).	
Phantom Reads	Definition 1.4.4
	When a transaction reads a set of records twice, but the sets of records are not equal as another transaction committed between the reads.

#### 1.4.5 Isolation levels

Serialisable	Definition 1.4.5
<i>Dirty Read</i>   <i>Non-repeatable Read</i>   <i>Phantom Read</i> Prevented   Prevented   Prevented	
Execution of transactions is can be serialized (it is equivalent to some sequential history of transactions).	
• In lock-based concurrency control locks are released at the end of a transaction, and range-locks are acquired for <code>SELECT ... FROM ... WHERE ... ;</code> to avoid <i>phantom reads</i> .	
• Prevents all 3 read phenomena and is the strongest isolation level.	
Repeatable Reads	Definition 1.4.6
<i>Dirty Read</i>   <i>Non-repeatable Read</i>   <i>Phantom Read</i> Prevented   Prevented   Allowed	
• Unlike <i>serialisable</i> Range locks are not used, only locks per-record.	
• Write skew can occur (when concurrent transactions write to the same table & column using data read from the table, resulting in a mix of both transactions)	
Read Committed	Definition 1.4.7
<i>Dirty Read</i>   <i>Non-repeatable Read</i>   <i>Phantom Read</i> Prevented   Allowed   Allowed	
Mutual exclusion is held for writes, but reads are only exclusive until the end of a <code>SELECT ... ;</code> statement, not until commit time.	
• In lock-based concurrency, write locks are held until commit, read locks released after select completed.	
Read Uncommitted	Definition 1.4.8
<i>Dirty Read</i>   <i>Non-repeatable Read</i>   <i>Phantom Read</i> Allowed   Allowed   Allowed	
The weakest isolation level and allows for all <i>read phenomena</i> .	

#### 1.4.6 Declarative Data Analysis

In order to make complex data management tools easier to use, a programmer describes the result they need declaratively, and the database system then plans the operations that must occur to provide the requested result.

This is present in almost all databases (e.g SQL)

# Chapter 2

## Relational Algebra

### 2.1 Relational Structures

#### 2.1.1 Preliminaries

##### Schema

##### Definition 2.1.1

A description of the database structure.

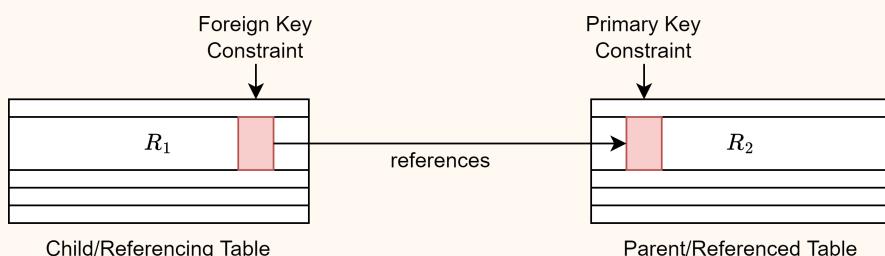
- Tables, names and types.

```
CREATE TABLE foo (bing INTEGER, zog TEXT, bar INTEGER);  
ALTER TABLE foo ADD CONSTRAINT foo_key UNIQUE(bing);
```

##### Foreign Key

##### Example Question 2.1.1

What is a foreign key constraint? Is it *like a pointer*?



It adds the invariant that there is a record referenced by the foreign key.

It is not really *like a pointer* as:

- Not in memory (e.g. on disk, different machine etc)
- No constant lookup (a pointer can be dereferenced in constant time, but looking up a key in a table is not necessarily)

Data structures used include:

Vector	Ordered collection of objects (same type)
Tuple	Ordered collection of objects (can be different types)
Bag	Unordered collection of objects (same type)
Set	Unordered collection of unique objects (same type)

## Relation

## Definition 2.1.2

An array representing an  $n$ -ary relation  $R$  with the properties:

- 1. Each row is an  $n$ -tuple of  $R$
- 2. Rows are unordered
- 3. All rows are unique / distinct
- 4. The order of columns corresponds to the ordering of the domains of  $R$
- 5. Each column is labelled

They are almost equivalent to sets tuples (but include labels).

Columns:	X-ray	Yankee	...	Zulu	Unique
	( $x_1$   $y_1$   ...   $z_1$ )				
	( $x_2$   $y_2$   ...   $z_2$ )				
	( $x_3$   $y_3$   ...   $z_3$ )				
	( $x_4$   $y_4$   ...   $z_4$ )				
	⋮	⋮	⋮	⋮	
	( $x_n$   $y_n$   ...   $z_n$ )				

Type  $(X, Y, \dots, Z)$

The minimal set of operators required for the relational algebra are:

Project    Select    Cross/Cartesian product    Union    Difference

Relational algebra is closed:

- Every operator outputs a relation
- Operators are unary or binary

## Query This!

## Example Question 2.1.2

Given the below structure, write a query.

**UNFINISHED!!!**

$\Pi_{title}(\sigma_{OrderItem.BookID=Book.BookID}(\sigma_{OrderedItem.OrderId = Order.OrderID}((\sigma_{Order.CustomerID=Customer.CustomerID}(\sigma_{customerID=}))$

```
SELECT Book.title
FROM (
  Customer NATURAL JOIN Order
) NATURAL JOIN OrderedItem
) NATURAL JOIN Book
```

Here a natural join is:

natural join( $R_1, R_2$ )  $\triangleq \sigma_{R_1.x_1=R_2.x_1 \wedge \dots \wedge R_1.x_n=R_2.x_n}(R_1 \times R_2)$  where the  $x$ s are in both tables

## Unique Addresses

## Example Question 2.1.3

**UNFINISHED!!!**

$\Pi_{Book.Author}(\sigma_{count=1}(\Gamma_{(Customer.ShippingAddress),(Book.Author,count)}(\Pi_{Book.Author, Customer.ShippingAddress}(\text{natural join}(Customer, Order))))$

```
SELECT Book.Author
FROM (
  SELECT Book.Author, Customer.ShippingAddress
  FROM (
    Customer NATURAL JOIN Order
```

```

    ) NATURAL JOIN OrderedItem
) NATURAL JOIN Book
)
GROUP BY Book.Author
WHERE COUNT(*) = 1;

```

## 2.1.2 Nomenclatures

<b>Expression</b>	A composition of operators
<b>Logical Plan/Plan</b>	An expression.
<b>Cardinality</b>	The number of tuples in a set.

## 2.2 Implementing Relational Algebra in C++

In order to implement relations we will make use of several containers from the STL (standard template library).

```

#include <set>
#include <array>
#include <string>
#include <tuple>
#include <variant>

using namespace std;

```

We will also make use of *variadic templates/parameter packs* to make our structures not only generic, but generic over  $n$  types.

```
template<typename... some_types>
```

We will also create an operator to inherit from for all operator types:

```
template <typename... types> struct Operator : public Relation<types...> {};
```

Finally when concatenating lists of types in templates, we will make use of the following:

```

// declare the empty struct used to bind types
template <typename, typename> struct ConcatStruct;

// Table both types, create a type alias within the scope of ConcatStruct that
// concatenates the lists of types
template <typename... First, typename... Second>
struct ConcatStruct<std::tuple<First...>, std::tuple<Second...>> {
    using type = std::tuple<First..., Second...>;
};

// expose the type alias outside of the scope of concatStruct
template <typename L, typename R>
using Concat = typename ConcatStruct<L, R>::type;

```

### 2.2.1 Relation

```

template <typename... types> struct Relation {
    // To allow relations to be composed, an output type is required
    using OutputType = tuple<types...>;

    set<tuple<types...>> data;           // table records
    array<string, sizeof...(types)> schema; // column names

    Relation(array<string, sizeof...(types)> schema, set<tuple<types...>> data)
        : schema(schema), data(data) {}
};

```

We can hence create a relation using the `Relation` constructor.

```
Relation<string, int, int> rel(
    {"Name", "Age", "Review"}, 
    {{ "Jim",     33,      3}, 
    { "Jay",     23,      5}, 
    {"Mick",    34,      4}} 
);
```

## 2.2.2 Project

$$\underbrace{\Pi_{a_1, \dots, a_n}}_{\text{columns}}(R)$$

A unary operator returning a relation containing only the columns projected  $(a_1, \dots, a_n)$ .

We can first create a projection to

```
template <typename InputOperator, typename... outputTypes>
struct Project : public Operator<outputTypes...> {
    // the single input
    InputOperator input;

    // a variant is a type safe union. It is either a function on rows, or a
    // mapping of columns
    variant<function<tuple<outputTypes...>(&InputOperator::OutputType)>,
        set<pair<string, string>>>
    projections;

    // Constructor for function application
    Project(InputOperator input,
            function<tuple<outputTypes...>(&InputOperator::OutputType)>
            projections)
        : input(input), projections(projections) {}

    // Constructor for column mapping
    Project(InputOperator input, set<pair<string, string>> projections)
        : input(input), projections(projections) {}
};
```

### SQL vs RA

### Extra Fun! 2.2.1

The default SQL projection does not return a set but rather a multiset / bag. In order to remove duplicates the `DISTINCT` keyword must be used.

## 2.2.3 Select

$$\sigma_{\text{predicate}}(R)$$

Produce a new relation of input tuples satisfying the predicate. Here we narrow this to a condition.

```
enum class Comparator { less, lessEqual, equal, greaterEqual, greater };

// user must explicitly set string as a column (less chance of mistake)
struct Column {
    string name;
    Column(string name) : name(name) {}
};

// type alias for comparable values
using Value = variant<string, int, float>;
```

```

struct Condition {
    Comparator compare;

    Column leftHandSide;
    variant<Column, Value> rightHandSide;

    Condition(Column leftHandSide, Comparator compare,
              variant<Column, Value> rightHandSide)
        : leftHandSide(leftHandSide), compare(compare),
          rightHandSide(rightHandSide) {}

};
```

#### Enums vs Enum classes

#### *Extra Fun! 2.2.2*

enum class	enum
Enumerations are in the scope of the class No implicit conversions.	Enumerations are in the same scope as the enum Implicit conversions to integers.
Enum classes are generally preferred over enums due to the above differences.	

#### 2.2.4 Cross Product / Cartesian

$$R_1 \times R_2$$

Creates a new schema concatenating the columns and with the cartesian product of records.

```

// Concat<> is used to concatenate the types from both input relations to
// produce a new schema
template <typename LeftInputOperator, typename RightInputOperator>
struct CrossProduct
    : public Operator<Concat<typename LeftInputOperator::OutputType,
                           typename RightInputOperator::OutputType>> {
// The input relations
LeftInputOperator leftInput;
RightInputOperator rightInput;

CrossProduct(LeftInputOperator leftInput, RightInputOperator rightInput)
    : leftInput(leftInput), rightInput(rightInput){};

};
```

#### 2.2.5 Union

$$R_1 \cup R_2$$

The union of both relations, duplicates are eliminated.

```

template <typename LeftInputOperator, typename RightInputOperator>
struct Union : public Operator<typename LeftInputOperator::outputType> {

    LeftInputOperator leftInput;
    RightInputOperator rightInput;

    Union(LeftInputOperator leftInput, RightInputOperator rightInput)
        : leftInput(leftInput), rightInput(rightInput){};

};
```

#### 2.2.6 Difference

$$R_1 - R_2$$

Get the set difference between two relations.

```

template <typename LeftInputOperator, typename RightInputOperator>
struct Difference : public Operator<typename LeftInputOperator::outputType> {
    LeftInputOperator leftInput;
    RightInputOperator rightInput;

    Difference(LeftInputOperator leftInput, RightInputOperator rightInput)
        : leftInput(leftInput), rightInput(rightInput){};

};

```

## 2.2.7 Group Aggregation

$$\Gamma_{(\text{grouping attributes}), (\text{aggregates})}(R)$$

- Records are grouped by equality on the *grouping attributes*
- A set of *aggregates* are produced (either a grouping attribute, the result of an aggregate function, or output attribute (e.g constants))

This is implemented by `GROUP BY` in SQL:

```

SELECT -- aggregates
FROM -- R
GROUP BY -- grouping attributes

// Aggregate functions to apply, 'agg' is for using groupAttributes
enum class AggregationFunction { min, max, sum, avg, count, agg };

template <typename InputOperator, typename... Output>
struct GroupedAggregation : public Operator<Output...> {
    InputOperator input;

    // the attributes to group by (column names)
    set<string> groupAttributes;

    // (column, aggregate function, new column name)
    set<tuple<string, AggregationFunction, string>> aggregations;

    GroupedAggregation(
        InputOperator input, set<string> groupAttributes,
        set<tuple<string, AggregationFunction, string>> aggregations)
        : input(input), groupAttributes(groupAttributes),
        aggregations(aggregations){};

};

```

## 2.2.8 Top-N

$$TopN_{(n, \text{attribute})}(R)$$

Get the top  $n$  records from a table, given the ordering of *attribute*

This is implemented with `LIMIT` and `ORDER BY` in SQL:

```

SELECT -- ...
FROM -- R
ORDER BY

// note that here we include N in the type (known at compile time), we could also
// take it as a parameter constructor (known at runtime)
template <typename InputOperator, size_t N>
struct TopN : public Operator<typename InputOperator::OutputType> {
    InputOperator input;
    string predicate;

```

```
TopN(InputOperator input, string predicate)
    : input(input), predicate(predicate){};

};
```

# Chapter 3

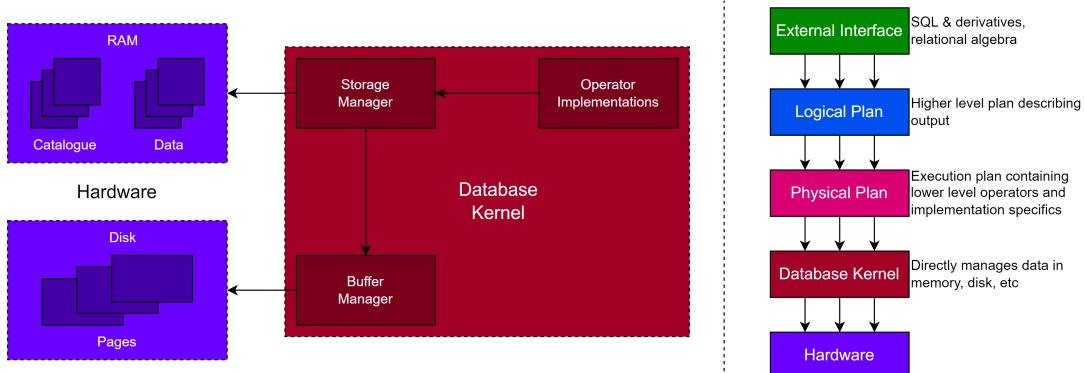
## Storage

### Great Exceptions!

### Extra Fun! 3.0.1

There are exceptions to many of the rules, implementation details discussed in this course. Most of the good (and bad) ideas considered here have been implemented several ways.

### 3.1 Database Management System Kernel



#### Database Kernel

#### Definition 3.1.1

The core of the database management system.

- Manages interaction with hardware (e.g I/O, memory management, operations)
- Library of functionality that implements physical plan & upwards.
- Provides an interface to access subsystems

Many often bypass the operating system to implement functionality usually associated with OS kernels.

## 3.2 Storage

### 3.2.1 Storage Manager

Multi-dimensional data must be stored in a 1-dimensional memory.

- Here we assume the tuples contain data types of a fixed size.
- Access latency of memory is determined by cache, hence locality is a key consideration.
- We need to consider the access pattern.
- Tables are externally represented as a set of tuples.
- We assume no concurrency for simplicity here.

The 60001 - Advanced Computer Architecture module by Prof Paul Kelly covers caches and access latency in great depth.

## Locality

## Definition 3.2.1

Average memory access latency is reduced using multiple levels of caches. These caches are designed to take advantage of locality in memory accesses within a program.

<b>Spatial</b>	Accessing nearby/-contiguous locations.	A cache miss on a word results in entire line (typically larger than a word) begin cached. Hardware prefetchers fetch lines adjacent to misses.
<b>Temporal</b>	Accessing the same location.	Lines stay until evicted due to capacity or flush, load-store queues effectively cache resent accesses.

## N-ary Storage

## Definition 3.2.2

Tuples are stored adjacently.



- Good spatial locality on access to all fields in a tuple.
- Works well for lookups and inserts (common in *OTP* where transactions typically run on recent data)

## Decomposed Storage

## Definition 3.2.3

Each field of the tuple is stored in a separate array.



- Good spatial locality when accessing one field of many tuples.
- Requires tuples to be reconstructed.
- Works well for scan-heavy queries (common in *OLAP* - aggregate, join and filtering)

## Delta/Main

## Definition 3.2.4

A hybrid of *n-ary* and *decomposed* storage.



- Complicates some operations (e.g. lookups)
- Regular migrations can reduce database availability at some points (lock up table to merge)
- Can be implemented as a pattern using two separate DBMS (transactional system and data warehouse).

## 3.2.2 Catalog

### Catalog

### Definition 3.2.5

Keeps track of database structure (tables, view, indexes etc) and metadata (e.g which tables are sorted, dense)

### Dense

### Definition 3.2.6

Records are both sorted and consecutive (e.g 3, 4, 5) in some field. Given fixed-size records and the minimum value, records can be looked up in constant time.

## 3.2.3 Disk Storage

### Buffer Manager

### Definition 3.2.7

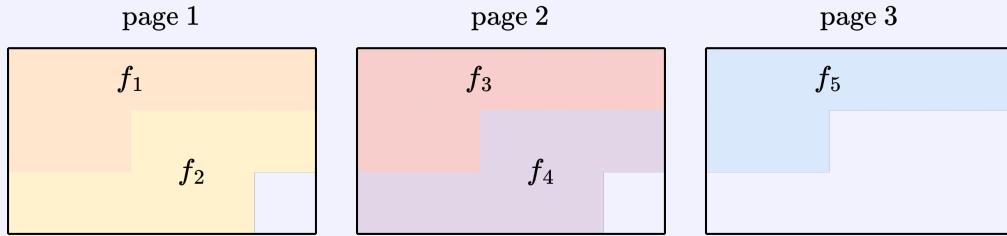
Manages disk-resident data and manages data transfer to *pages* in memory.

- Unstructured files → structured tables
- Ensures fixed size for files.
- Safely writes data to disk when necessary (to ensure durability).

## Unspanned Pages

## Definition 3.2.8

Records only allocated on the page is there is space.

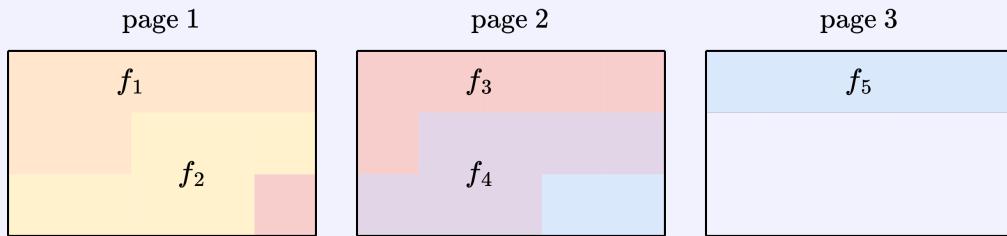


- Space wasted (larger as tuple size increases)
- If the record size > page size, it is not possible to use this strategy
- Assuming there is a known fill factor (number of tuples allocated to each page) we can get fast random lookup for the page a variable size tuple is on.
- If records are variable size, no constant time random access within a page.

## Spanned Pages

## Definition 3.2.9

Records placed across page boundaries.

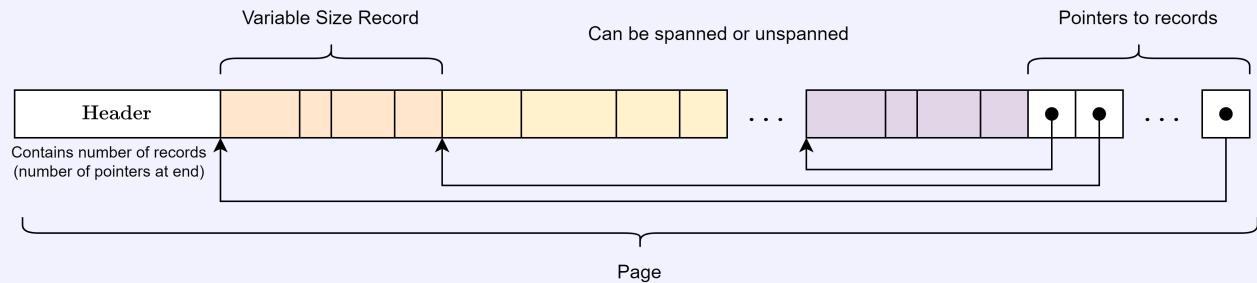


- Minimises wasted space
- Supports very large record sizes (larger than a page)
- Complex to implement, and reduced random access performance (with variable size tuples we cannot determine the page a tuple is on in constant time)
- If records are variable size, no constant time random access within a page.

## Slotted Pages

## Definition 3.2.10

To allow faster/constant time lookup for variable size records.



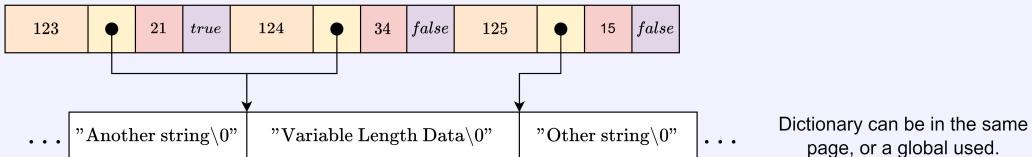
Header stores number of records, index of record used to look-up pointers at the end of the page which are dereferenced to get the record.

Rather than store data (particularly variable-size) in-place it is allocated elsewhere, and a pointer used.

#### Variable Size Records

123	"Variable Length Data\0"	21	true	124	"Variable Length Data\0"	34	false	125	"Other string\0"	15	false
In-place, variable size data											

Data stored out-of-place, using pointers (fixed size)



Dictionary can be in the same page, or a global used.

- Can eliminate duplication (duplicate attributes point to the same data)
- Need to be careful about managing space (e.g periodically removing unused dictionary entries / garbage collection)
- Can reduce spatial locality (record points to non-adjacent dictionary entry), but can (sometimes) improve temporal (same dictionary value accessed many times from many records)

**In-Page** Dictionary accesses from within the page do not require other pages to be loaded.

Globally more duplicates may exist & fewer records can be held per page.

**Global** A large global dictionary is used (access from other pages require loading).

#### Lookups

#### Example Question 3.2.1

Access latencies to elements in a table are modelled as follows:

- $n$  Access a (32 bit) word for the first time
- $m$  Access an adjacent word to the last accessed word where  $n > m > p$
- $p$  Access a previously accessed value

**UNFINISHED!!!**

### 3.3 Example Sketch Implementation

**UNFINISHED!!!**

# Chapter 4

# Algorithms and Indices

## 4.1 Sorting Algorithms

- 4.1.1 Quicksort
- 4.1.2 Merge Sort
- 4.1.3 Tim Sort
- 4.1.4 Radix Sort
- 4.1.5 Top-N with Heaps

**UNFINISHED!!!**

## 4.2 Joins

- 4.2.1 Database Normalisation

**UNFINISHED!!!**

### 4.2.2 Join Types

Normalised databases naturally require joins to re-compose data.

We would be honoured if you would join us...

Example Question 4.2.1

Provide some examples of types of queries that would require a join.

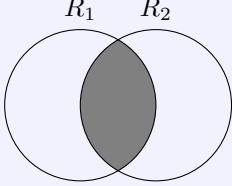
**UNFINISHED!!!**

#### Join

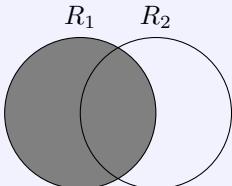
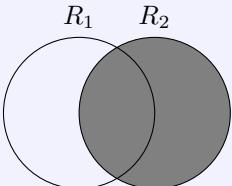
#### Definition 4.2.1

A join is a cross product with selection using data from both relations ( $\sigma_{p(R_A.x, R_B.y)}(R_A \times R_B)$ ).

## Inner Joins

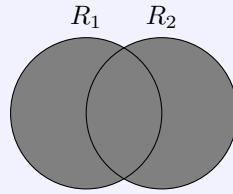
Inner Join	Definition 4.2.2	Natural Join	Definition 4.2.3
A join only returning rows from both tables which satisfy a predicate/condition.			Joining two tables with an implicit join clause (join on equality on a column present in both tables)
			$R_1 \bowtie R_2$
<b>FROM</b> R1 <b>NATURAL JOIN</b> R2 <b>FROM</b> R1 <b>JOIN</b> R2 <b>USING</b> (id)			
Theta Join	Definition 4.2.4	Equi Join	Definition 4.2.5
Joining two tables based on a condition/predicate $\theta$ .			A <b>theta join</b> with a single equivalence condition. A <b>natural join</b> is an implicit <b>equi join</b> .
$R_1 \stackrel{\theta}{\bowtie} R_2$ <b>FROM</b> R1, R2 <b>WHERE</b> theta(R1, R2) <b>FROM</b> R1 <b>JOIN</b> R2 <b>ON</b> theta(R1, R2)			$R_1 \bowtie_{R_1.x=R_2.x} R_2$ <b>FROM</b> R1, R2 <b>WHERE</b> R1.x = R2.x
Cross Join	Definition 4.2.6	Anti Join	Definition 4.2.7
Just cartesian product with no selection.			A <b>theta join</b> using an inequality predicate
$R_1 \times R_2$ <b>FROM</b> R1, R2 <b>FROM</b> R1 <b>CROSS JOIN</b> R2			$R_1 \bowtie_{R_1.x <> R_2.x} R_2$ <b>FROM</b> R1 <b>JOIN</b> R2 <b>ON</b> R1.x <> R2.x

## Outer Joins

Left Join	Definition 4.2.8	Right Join	Definition 4.2.9
$R_1 \overset{L}{\bowtie} R_2$ >Returns all rows of $R_1$ even if no rows in $R_2$ match (in which case columns are <b>NULL</b> ).			$R_1 \overset{R}{\bowtie} R_2$ >Returns all rows of $R_2$ even if no rows in $R_1$ match (in which case columns are <b>NULL</b> ).
			
<b>FROM</b> R1 <b>LEFT JOIN</b> R2 <b>ON</b> ...			<b>FROM</b> R1 <b>RIGHT JOIN</b> R2 <b>ON</b> ...

$$R_1 \overset{O}{\bowtie} R_2 \equiv R_1 \overset{L}{\bowtie} R_2 \cup R_1 \overset{R}{\bowtie} R_2$$

Returns all rows from all tables matching, with rows from either  $R_1$  or  $R_2$  that do not have match associated with **NULL** columns from the other table.



```
FROM R1 FULL OUTER JOIN R2 ON ...
FROM R1 FULL JOIN R2 ON ...
FROM (SELECT * FROM R1 LEFT JOIN R2 ON ... UNION SELECT * FROM R1 RIGHT JOIN R2 ON ...)
```

### Which imposter?

### Example Question 4.2.2

Which of the following are joins?

- |  |   |
|--|---|
| 1. <code>SELECT R.r, S.s<br/>FROM R, S<br/>WHERE R.id = S.id;</code> | 3. <code>SELECT R.r<br/>FROM R, S<br/>WHERE R.id = S.id;</code>         |
| 2. <code>SELECT R.r, S.s<br/>FROM R, S<br/>WHERE R.r = R.id</code>   | 4. <code>SELECT R.r<br/>FROM R, S<br/>WHERE R.r = "some string";</code> |

1. **Join** (Selects on both  $R$  and  $S$ )
2. **Not a Join** (Only selects on  $R$ )
3. **Join** (The  $\sigma$  selection is on  $R$  and  $S$ , so a join even if only  $R$  is projected)
4. **Not a Join** (Only selects on  $R$ )

### 4.2.3 Join Implementations

The following join implementations are written in C++20

```
#include <algorithm>
#include <iostream>
#include <tuple>
#include <unordered_map> // using contains from cpp20
#include <utility>
#include <vector>

using namespace std;

template <typename... types> using Table = vector<tuple<types...>>;
```

Compile with `g++ -std=c++2a joins.cc`, the following main can be used for testing:

```
int main() {
    vector<tuple<int, char, int>> table1{
        {1, 'a', 21}, {1, 'b', 34}, {2, 'c', 23}};
    vector<tuple<char, int>> table2{{'a', 21}, {'b', 34}, {'c', 6}};

    auto tableResult = sort_merge_join<2, 1>(table1, table2);
```

```

print_table(table1);
print_table(table2);
print_table(tableResult);
}

#include "print_table.cc" after the using Table = ... definition for easy printing.

```

#### 4.2.4 Nested Loop Join

We can implement a basic join naively using nested loops.

```

template <size_t leftCol, size_t rightCol, typename... TypesOne, typename... TypesTwo>
Table<TypesOne..., TypesTwo...> nest_loop_join(Table<TypesOne...> &left, Table<TypesTwo...> &right) {
    Table<TypesOne..., TypesTwo...> result;
    for (auto &leftElem : left) for (auto &rightElem : right) {
        if (get<leftCol>(leftElem) == get<rightCol>(rightElem)) {
            result.push_back(tuple_cat(leftElem, rightElem));
        }
    }
    return result;
}

```

$$\text{Time Complexity} = \begin{cases} \frac{\Theta(|left| \times |right|)}{2} & \text{If elements unique} \\ \Theta(|left| \times |right|) & \text{otherwise} \end{cases}$$

<b>Simple</b>	Easy to reason about (memory accesses & complexity)
<b>Trivially Parallel</b>	Loop iterations are not dependent, so can be parallelised.
<b>Sequential I/O</b>	Access is done in the order of the tables storage (sequential access better for both memory & disk)

**Performance** Linear time complexity.

#### 4.2.5 Sort Merge Join

If we assume both tables are sorted, and values (being joined on) are unique.

- Two cursors (one per table)
- Advance cursors in order, if the value on the left exceeds the right there can be no joins for the left row (and vice versa).

```

template <size_t leftCol, size_t rightCol, typename... TypesOne,
          typename... TypesTwo>
Table<TypesOne..., TypesTwo...> sort_merge_join(const Table<TypesOne...> &leftT,
                                                const Table<TypesTwo...> &rightT) {

    Table<TypesOne..., TypesTwo...> result;

    // copy tables and sort (required for interface, but could be omitted with assumption)
    auto left = leftT;
    auto right = rightT;
    sort(left.begin(), left.end(), [](auto const &a, auto const &b) {
        return get<leftCol>(a) < get<leftCol>(b);
    });
    sort(right.begin(), right.end(), [](auto const &a, auto const &b) {
        return get<rightCol>(a) < get<rightCol>(b);
    });

```

```

auto leftIndex = 0;
auto rightIndex = 0;

while (leftIndex < left.size() && rightIndex < right.size()) {
    auto leftElem = left[leftIndex];
    auto rightElem = right[rightIndex];

    if (get<leftCol>(leftElem) < get<rightCol>(rightElem)) {
        leftIndex++;
    } else if (get<leftCol>(leftElem) > get<rightCol>(rightElem)) {
        rightIndex++;
    } else {
        result.emplace_back(tuple_cat(leftElem, rightElem));
        leftIndex++;
        rightIndex++;
    }
}

return result;
}

```

$$\begin{aligned}
\text{Time Complexity} &= \Theta(\text{sort}(left)) + \Theta(\text{sort}(right)) + \Theta(\text{merge}) \\
&= \Theta(|left| \times \log |left| + |right| \times \log |right| + |left| + |right|)
\end{aligned}$$

**Sequential I/O** In the merge phase

**Inequality** Works for joins using  $<$  and  $>$  instead of just *equi-joins*.

**Tricky to Parallelize** Sorts can be somewhat parallelised, but merge is sequential.

#### 4.2.6 Hash Join

For *equi joins* we can insert one table into a hash table, then iterate over the second (assumed constant time lookup in hashtable).

Below we have used the standard template library's `unordered_map`

```

template <size_t leftCol, size_t rightCol, typename... TypesOne, typename... TypesTwo>
Table<TypesOne..., TypesTwo...> hash_join(const Table<TypesOne...> &left,
                                              const Table<TypesTwo...> &right) {
    Table<TypesOne..., TypesTwo...> result;
    using leftColType = typename tuple_element<leftCol, tuple<TypesOne...>>::type;

    // Build Phase - create has hashtable of one table.
    // we should ideally choose the smallest table here -> smallest hashmap
    unordered_map<leftColType, const tuple<TypesOne...>> leftContents(
        left.size());

    // Inserting pointers to avoid overhead of cloning tuples
    for (const tuple<TypesOne...> &elem : left) {
        leftContents.insert(make_pair(get<leftCol>(elem), &elem));
    }

    // Probing phase - find matching values
    for (auto &elem : right) {
        if (leftContents.contains(get<rightCol>(elem))) {
            result.emplace_back(tuple_cat(*leftContents[get<rightCol>(elem)], elem));
        }
    }
}

```

```

    }
    return result;
}

 $\Theta(|build| + |probe|)$  best case
 $O(|build| \times |probe|)$  worst case

```

- The probing phase can be easily parallelised (hashtable is unchanged), however the build side is tricky to parallelise efficiently.

<b>Time Complexity Hashing</b>	(Assuming the lookup is constant time). Need to avoid collisions, keep time calculating hash low, and be applicable to many data types.
--------------------------------	--

<b>Space Complexity</b>	Requires building a hashtable structure (assuming the table was not stored as this already). Best when one relation is much smaller than the other (use smallest).
<b>Expensive Hashing</b>	Some good hashing algorithms are expensive (potentially as many cycles as multiple data accesses).

## Bucket Based Hashmaps

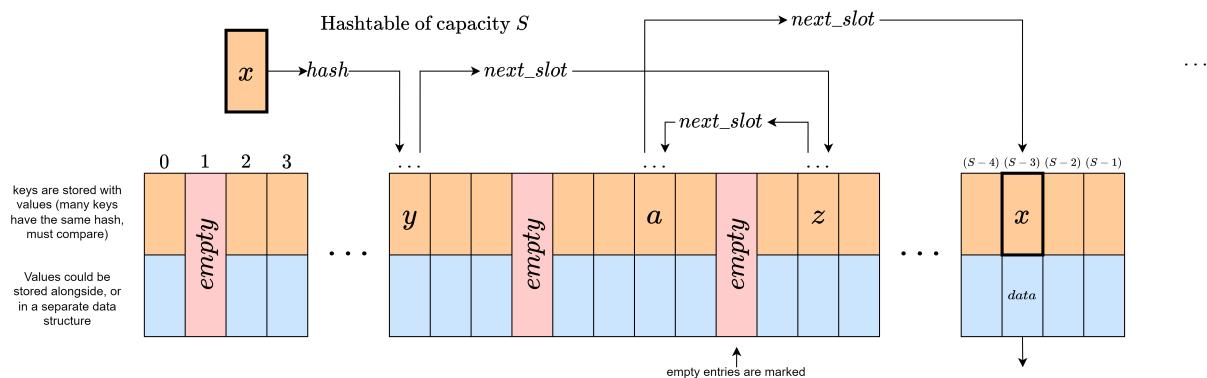
### Extra Fun! 4.2.1

Many hashmaps are implemented as a table of buckets (linked lists of conflicting values).

- Called bucket-chaining/open addressing
- Poor lookup performance.
- Good insert performance (can prepend to bucket linked list on conflict).

## 4.3 Hash Tables

### 4.3.1 Probing Hashmap



We can define the hash function using a `struct` as follows:

```

template <typename T> struct Probe {
    virtual void hash(T data, size_t indexSize) = 0;
    virtual size_t next() = 0;
};

```

Requirement	<b>Pure</b>	no state/call with same value → same hash
Requirement	<b>Known Co-Domain</b>	Known range of values (co-domain also known as image/range).
Nicety	<b>Contiguous Co-Domain</b>	No gaps in range of output means few gaps holes in the table.
Nicety	<b>Uniform</b>	All hash values in the range are equally likely.

<b>MD5</b>	Encodes any length string as a 128-bit hash.
<b>Modulo-Division</b>	Very simple and fast.
<b>MurmurHash</b>	A fast, non-cryptographic hash (on github).
<b>CRC32</b>	Cyclic Redundancy Check (common, non-cryptographic) and with hardware support on some systems (also see usage of <a href="#">PCLMULQDQ</a> on intel for acceleration here)

**Hash it out****Example Question 4.3.1**

Write a basic Modulo-Division hash using the interface above provided. Take the modulus as a template parameter.

```
template <size_t MODULUS> size_t modulusHash(int data) {
    return static_cast<size_t>(data) % MODULUS;
}
```

When different keys have the same hash a *conflict* occurs. A strategy is required to select the next slot to probe (the `nextSlot` function).

- We want locality (when detecting a conflict, the real key is close/same page/line)
- Very high locality will result in parts of the hash table being saturated, and long probe chains.
- We want to avoid leaving holes (may be used by hash function, but if the probing function never accesses, they are likely to never be used)

**Linear Probing**

Add some DISTANCE to the probe position, wrap around at the end of the buffer.

```
template <typename K> struct LinearProbe : public Probe<K> {
    LinearProbe(std::function<size_t(K)> hash) : _hash(hash) {}

    void hash(K data, size_t indexSize) override {
        _indexSize = indexSize;
        _position = _hash(data) % indexSize;
    }

    size_t next() override {
        auto oldPosition = _position;
        _position = (_position + 1) % _indexSize;
        return oldPosition;
    }

private:
    std::function<size_t(K)> _hash;
    size_t _position;
    size_t _indexSize;
};
```

**Simple** Easy to reason about memory access pattern.  
**Locality** Can alter DISTANCE to place values as *adjacently* as we need.

**Long Probe-Chains** From too much locality on adversarial input data (can input data to the table to create worse case conflicts (and hence probe chain length) scenario)

**Quadratic Probing**

$$P, P + 1^2, P + 2^2, P + 3^2, \dots, P + n^2, \dots$$

- Wrap around end of table.
- Variants exist (still use power of 2 but can include linear and constant term)

```
template <typename K> struct QuadraticProbe : public Probe<K> {
    QuadraticProbe(std::function<size_t(K)> hash) : _hash(hash) {}

    void hash(K data, size_t indexSize) override {
        _indexSize = indexSize;
        _firstPosition = _hash(data) % indexSize;
        _step = 0;
    }

    size_t next() override {
        auto newPosition = _firstPosition + _step * _step;
        _step++;
        return newPosition;
    }

private:
    std::function<size_t(K)> _hash;
    size_t _firstPosition;
    size_t _step;
    size_t _indexSize;
};
```

**Simple** Easy to reason about memory access pattern.  
**Locality** for first probes is good.

**Conflicts** Experiences conflicts in first probes where it is similar to linear.

## Rehashing

In order to distribute nodes uniformly, use a hash function to hash a conflicting position to find the next one.

```
template <typename K> struct ReHashProbe : public Probe<K> {
    ReHashProbe(std::function<size_t(K)> hash,
                std::function<size_t(size_t)> rehash)
        : _hash(hash), _rehash(rehash) {}

    void hash(K data, size_t indexSize) override {
        _indexSize = indexSize;
        _current = _hash(data) % indexSize;
    }

    size_t next() override {
        auto old = _current;
        _current = _rehash(_current) % _indexSize;
        cout << "rehashing to " << _current << endl;
        return old;
    }

private:
    std::function<size_t(K)> _hash;
    std::function<size_t(size_t)> _rehash;
    size_t _current;
    size_t _indexSize;
};
```

**Simple** To implement

**Reuse** Can potentially reuse the hashing function.

**Locality** is poor as probes distributed uniformly.

**Conflict Probability** is constant (every probe may conflict with another element).

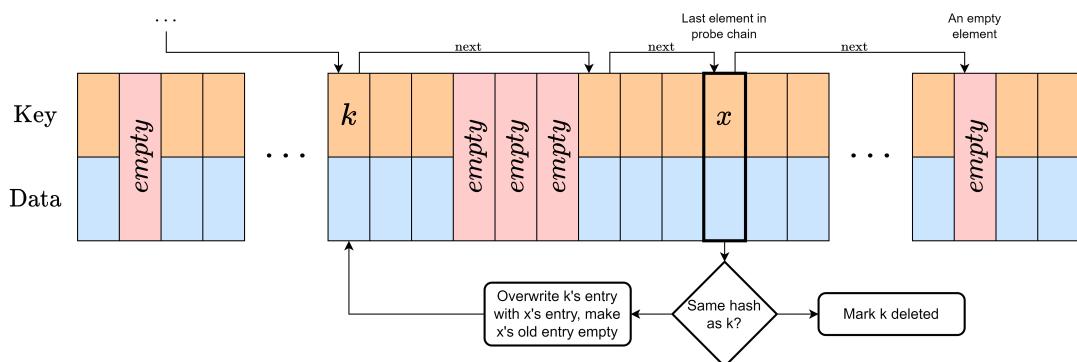
## Resizing

For the example above we have considered fixed-size hashmap.

- Hashtables are typically overallocated by factor 2 (twice as many slots as expected input tuples).
- Table can be resized once it is larger than some capacity (will change hash of values, so must effectively rebuild hashmap)
- When determining cost we amortise (spread cost of resize over all inserts & (for this module) assume this cost is constant per insert).

For this reason, hash-joins (using hash tables) are best when one of the joined relations is much smaller than the other.

## Deleting with Markers



An implementation is included below:

### 4.3.2 Basic Hash Table Implementation

Contribute!

Extra Fun! 4.3.2

This basic implementation can be improved!

- Resize functionality
- Use structs for entries, rather than tuples
- Construct probers local to methods
- Provide prober type in template and construct, rather than taking constructor parameter

```
#include <functional>
#include <iostream>
#include <optional>
#include <tuple>
#include <vector>

using namespace std;

// Produce hash from given data type
template <typename T> struct Probe {
```

```

virtual size_t hash(T data, size_t indexSize) = 0;
virtual size_t pureHash(T data) = 0;
virtual size_t next() = 0;
};

// class needs declaration of friend operator<< overload. But operator<<
// overload needs declaration of class, hence these declarations
template <typename K, typename V> class HashTable;
template <typename K, typename V>
ostream &operator<<(ostream &, const HashTable<K, V> &);

// a simple fixed-size hash-table for testing hashing and probing functions
template <typename K, typename V> class HashTable {
public:
    HashTable(Probe<K> &prober, const size_t initial_size = 50)
        : _slots(initial_size), _prober(prober){};

    // Attempt to insert a value, will return true if inserted, false if the key
// already existed in the table.
    bool insert(K key, V value) {
        // Start the prober (it hashes, first next is the first position)
        auto firstHash = _prober.hash(key, _slots.size());

        auto position = firstHash;
        auto slot = _slots[position];

        while (slot.has_value()) {
            auto &slotValue = slot.value();
            if (get<0>(slotValue) && get<1>(slotValue) == key) {
                // slot is present, and contains the key we want to insert (fail)
                return false;
            } else if (!get<0>(slotValue) &&
                      _prober.pureHash(get<1>(slotValue)) == firstHash) {
                // slot is not present, but is part of probe chain (fill with our value)
                // insert here (break loop)
                cout << "already in map" << endl;
                break;
            }
        }

        position = _prober.next();
        slot = _slots[position];
    }

    _slots[position] = make_tuple(true, key, value);
    return true;
}

// Search the table for the value, returning an optional of the result
optional<V> find(K key) {
    auto position = _prober.hash(key, _slots.size());
    auto slot = _slots[position];

    while (slot.has_value()) {
        auto slotValue = slot.value();

        if (get<1>(slotValue) == key) {
            if (get<0>(slotValue)) {
                return optional<V>(get<2>(slotValue));
            } else {
                // is in the map but deleted (cannot be anywhere else)
            }
        }
    }
}

```

```

        return optional<V>();
    }
}
position = _prober.next();
}
return optional<V>();
}

bool remove(K key) {
    auto firstHash = _prober.hash(key, _slots.size());
    auto position = firstHash;
    auto slot = _slots[position];

    auto lastpos = [this](size_t position) {
        auto nextPosition = _prober.next();
        while (_slots[nextPosition].has_value()) {
            position = nextPosition;
            nextPosition = _prober.next();
        }
        return position;
    };

    while (slot.has_value()) {
        auto slotValue = slot.value();

        if (get<1>(slotValue) == key) {
            if (get<0>(slotValue)) {
                // now get last tuple position in probe chain

                auto endpos = lastpos(position);

                if (endpos != position &&
                    _prober.pureHash(get<1>(_slots[endpos].value())) == firstHash) {
                    _slots[position] = _slots[endpos];
                    _slots[endpos] = optional<tuple<bool, K, V>>();
                } else {
                    // either pos is the endpos, or we could not find another element in
                    // the same probe chain. so just mark deleted.
                    _slots[position] = optional<tuple<bool, K, V>>(
                        make_tuple(false, get<1>(slotValue), get<2>(slotValue)));
                }
                return true;
            } else {
                // was already deleted
                return false;
            }
        }
        position = _prober.next();
        slot = _slots[position];
    }
    return false;
}

friend ostream &operator<<<K, V>(ostream &, const HashTable<K, V> &);
}

private:
    vector<optional<tuple<bool, K, V>>> _slots;
    Probe<K> &_prober;
};

```

```

template <typename K, typename V>
ostream &operator<<(ostream &os, const HashTable<K, V> &hashTable) {
    os << "Hash Table (Capacity " << hashTable._slots.size() << "):" << endl;
    for (size_t i = 0; i < hashTable._slots.size(); i++) {
        auto &elem = hashTable._slots[i];
        os << i << ": ";
        if (elem.has_value()) {
            os << "k: " << get<1>(elem.value()) << " v: " << get<2>(elem.value());
            if (!get<0>(elem.value())) {
                os << " (deleted)";
            }
        } else {
            os << "empty";
        }
        os << endl;
    }
    return os;
}

template <typename K> struct LinearProbe : public Probe<K> {
    LinearProbe(std::function<size_t(K)> hash) : _hash(hash) {}

    size_t hash(K data, size_t indexSize) override {
        _indexSize = indexSize;
        _position = pureHash(data);
        return _position;
    };

    size_t pureHash(K data) override { return _hash(data) % _indexSize; }

    size_t next() override {
        _position = (_position + 1) % _indexSize;
        return _position;
    };

private:
    std::function<size_t(K)> _hash;
    size_t _position;
    size_t _indexSize;
};

template <typename K> struct QuadraticProbe : public Probe<K> {
    QuadraticProbe(std::function<size_t(K)> hash) : _hash(hash) {}

    size_t hash(K data, size_t indexSize) override {
        _indexSize = indexSize;
        _firstPosition = pureHash(data);
        _step = 0;
        return _firstPosition;
    };

    size_t pureHash(K data) override { return _hash(data) % _indexSize; }

    size_t next() override {
        _step++;
        return _firstPosition + _step * _step;
    };

private:
    std::function<size_t(K)> _hash;
};

```

```

size_t _firstPosition;
size_t _step;
size_t _indexSize;
};

template <typename K> struct ReHashProbe : public Probe<K> {
    ReHashProbe(std::function<size_t(K)> hash,
                std::function<size_t(size_t)> rehash)
        : _hash(hash), _rehash(rehash) {}

    size_t hash(K data, size_t indexSize) override {
        _indexSize = indexSize;
        _current = pureHash(data);
        return _current;
    };

    size_t pureHash(K data) override { return _hash(data) % _indexSize; }

    size_t next() override {
        _current = _rehash(_current) % _indexSize;
        return _current;
    };
}

private:
    std::function<size_t(K)> _hash;
    std::function<size_t(size_t)> _rehash;
    size_t _current;
    size_t _indexSize;
};

size_t intIdHash(int data) { return static_cast<size_t>(data); }

template <size_t MODULUS> size_t modulusHash(int data) {
    return static_cast<size_t>(data) % MODULUS;
}

size_t basicRehash(size_t data) { return data * 13; }

int main() {
    // auto probe = ReHashProbe<int>(intIdHash, basicRehash);
    // auto probe = QuadraticProbe<int>(intIdHash);
    auto probe = LinearProbe<int>(intIdHash);

    auto table = HashTable<int, bool>(probe, 10);

    table.insert(3, true);
    table.insert(13, true);
    table.insert(23, true);
    table.insert(2, true);
    table.insert(22, true);
    cout << table << endl;

    table.remove(13);
    cout << table << endl;

    table.insert(13, false);
    cout << table << endl;
}

```

### 4.3.3 Partitioning

Sequential accesses are cheaper than random accesses, as they can access the same page in memory & thus share the cost of the initially expensive cold access.

$$c = \text{cost of page-in}$$

$$\frac{n}{\text{pagesize}_{OS}} \times c = \text{cost of sequentially accessing } n \text{ elements}$$

$$\frac{c}{\text{pagesize}_{OS}} = \text{cost of one access}$$

In order to reduce the cost of accessing some data we can:

- Increase the page size (huge pages).
- Make the access pattern *more* sequential.

Assuming a hashtable does not fit in memory/buffer page cache, we can reduce costs from page-misses by paying less for a partitioning pass.

**UNFINISHED!!!**

### 4.3.4 Indexing

We can use a secondary store of redundant data to speed up queries.

- Denormalised (redundant) data is controlled by the DBMS.
- Can be created or removed without affecting the system (other than performance & storage space).
- Semantically invisible to the user (cannot change semantics of queries).
- Can be used to speed up data access of some queries (e.g avoiding having to build a hashtable in hash join as it is already available).
- Occupy potentially considerable space.
- Must be maintained under updates.
- Must be considered by query optimiser.

#### Clustered/Primary Index

#### Definition 4.3.1

An index storing all tuples of a table.

- Only one per table
- Can use more space than the table being indexed
- No redundant data / no duplicates within the index (only one copy for each tuple is indexed) (no consistency issues)

#### Unclustered/Secondary Index

#### Definition 4.3.2

Used to store pointers to tuples of a table.

- No limit on number of indexes
- Does not replicate data (the tuples pointed to in the table), but may replicate pointers (multiple pointers in index to the same tuple in the table) (some consistency issues)

### SQL Indexes

ANSI SQL supports the creation & destruction of indexes by the user.

```
CREATE INDEX index_name ON table_name (column_1, column2, ...);  
DROP INDEX index_name;
```

- Unclear what type of index is created
- No control over parameters (e.g hash table size)

The standard has been extended by SQL implementations to allow for finer control.

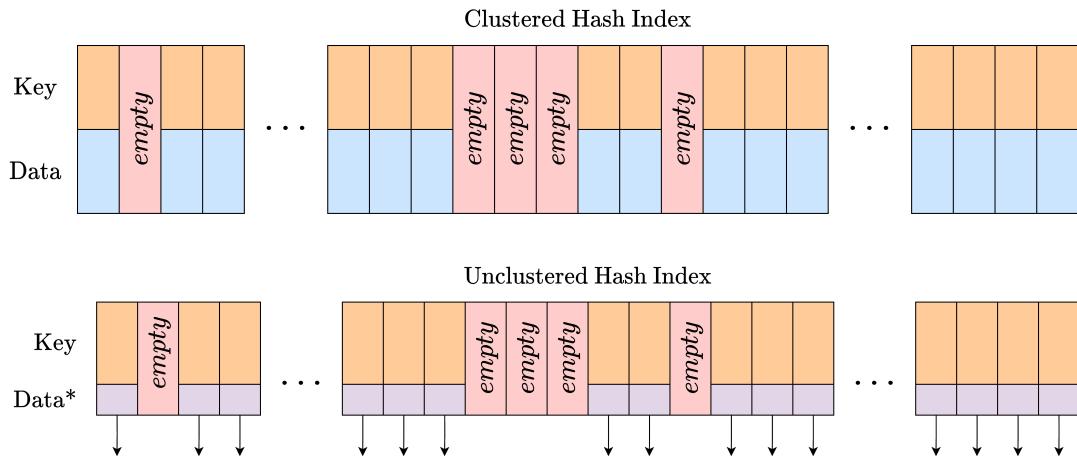
Among other DBMS, Postgres supports many types of index (documentation here)

```
/* By default CREATE INDEX uses a B-Tree */
CREATE INDEX name ON table USING HASH (column);

/* It is even possible to only index certain parts of a table using a WHERE clause */
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
WHERE NOT (client_ip > inet '192.168.100.0' AND
           client_ip < inet '192.168.100.255');
```

### 4.3.5 Hash Indexes

An index backed by a hash table.



Persistent hash tables may grow very large (overallocate) and need to be rebuilt to grow (can cause unexpected spike when an insert causes a rebuild).

Aside from the normal pros/cons of hash tables in general:

<b>Hash-Joins &amp; Aggregation</b>	Perform well and remove build phase (provided they index on the columns joining).
<b>Equality Selection</b>	Can reduce number of candidate columns if not all columns are indexed
<code>SELECT * FROM table_name WHERE column1 = some_value;</code>	

**Limited Applicability** Not useful for queries not using equality.

### 4.3.6 Bitmap Indexing

#### Bit Vector

#### Definition 4.3.3

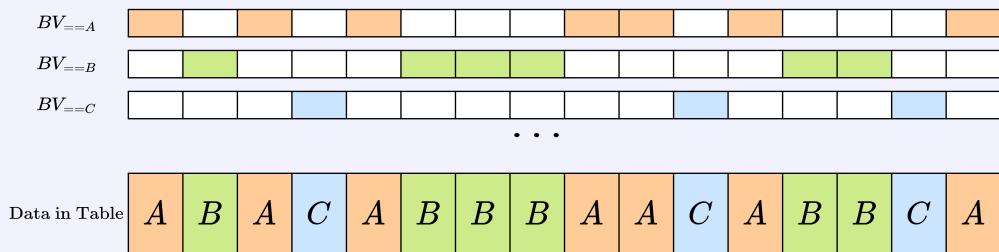
A sequence of 1 bit boolean values indicating some condition holds for indexes of another sequence.

$$BV_{==3}([1, 2, 5, 6, 3, 2, 3, 4]) = [0, 0, 0, 0, 1, 0, 1, 0]$$

- Memory is byte addressable, and registers typically word-size (usually 32/64 bits).
- Some useful instructions (and compiler intrinsics) can be used.
- Can use SIMD instructions to operate on sections of a bitvector in parallel without using multithreading.

## Bitmap Index

## Definition 4.3.4



A collection of bitvectors on a column (each for a distinct value in the column).

- Need one bitvector per distinct value in the column
- Bitvectors usually disjoint (column can only be one distinct value at one time)

$$\text{size}(\text{rows}, \text{distinct\_values}) = \frac{\text{rows} \times \text{distinct\_values}}{8} \text{ bytes}$$

On some systems we can create an index of arbitrary predicates, and to scan multiple bitmaps (using boolean operators on them).

The CPU operates in word size chunks of the bitvector. Hence we can easily check if all bits in a word size chunk (e.g 32 bits) are zero. We only need to iterate through this chunk if the chunk is non-zero.

```
#include <cstdint>
#include <iostream>
#include <vector>

using namespace std;

// scans a vector of 32 bit ints:
// - indexes each integer from LSB(0) to MSB(31)
// - does not consider endian-ness
// 100... 100... <=> [1,1]
vector<size_t> scan_bitmap(const vector<uint32_t> &bitvector) {
    vector<size_t> positions;
    size_t index = 0;
    for (auto elem : bitvector) {
        for (size_t small_index = 0; elem; small_index++, elem >>= 1) {
            if (elem & 1) {
                positions.push_back(index + small_index);
            }
        }
        index += 32;
    }
    return positions;
}
```

**Bandwidth** Can scan a column with reduce memory bandwidth (e.g integers → bitmap index is 32 times less).

**Flexibility** Can often use arbitrary predicates (e.g  $< x$ ) to either turn a filter into a bitmap scan, or reduce time to scan (if  $x < y$  an index  $< x$  and help with a  $< y$  filter).

## Binned Bitmaps

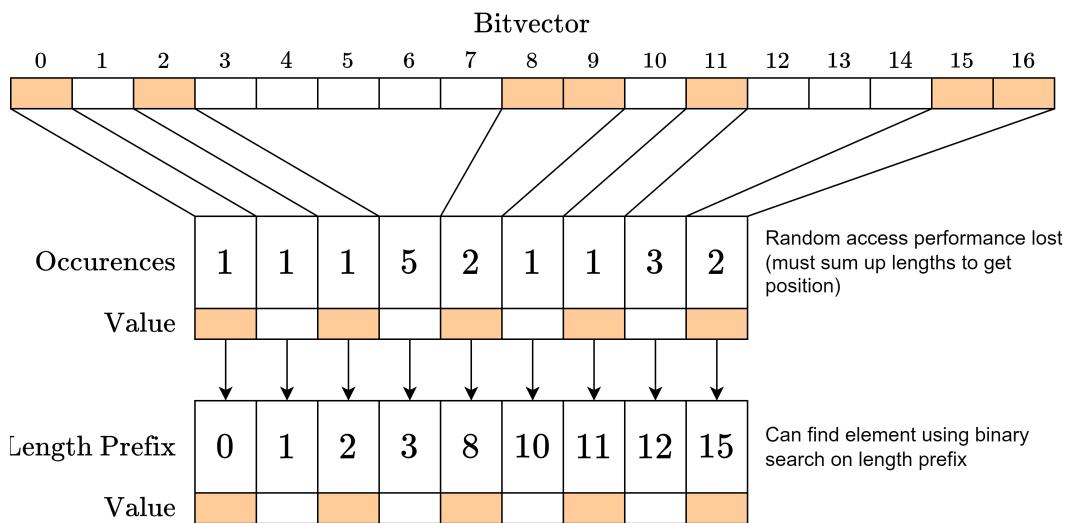
When there is a high number of distinct values, but we do not want many bitvectors, we can create several bitvectors covering ranges of values.

- The bitvectors ranges need to cover the entire domain.
- Smaller range → more precise and more useful for queries concerning data in that range, at the cost of more space used (more bitvectors)
- Not all ranges need to be the same size, we can use the distribution of values to determine the ranges of the bins.

The *false-positive rate* given a filter for  $z$ , and a bin of range  $(x, y)$  where  $x < z < y$ , what proportion of the 1s in the bitvector are not for the value  $z$ .

Equi-Width	Definition 4.3.5	Height Binning	Definition 4.3.6
$width = \frac{\max(column) - \min(column)}{number\_bins}$ Split range into several equal size bins. Useful for uniformly distributed (or near) data.		All bins should contain the same number of values. <ul style="list-style-type: none"> <li>• Construction difficult (usually sort, determine quartiles on a sample)</li> <li>• False-positive rate is value independent</li> <li>• As table changes, may need to re-bin.</li> </ul>	

### Run Length Encoding

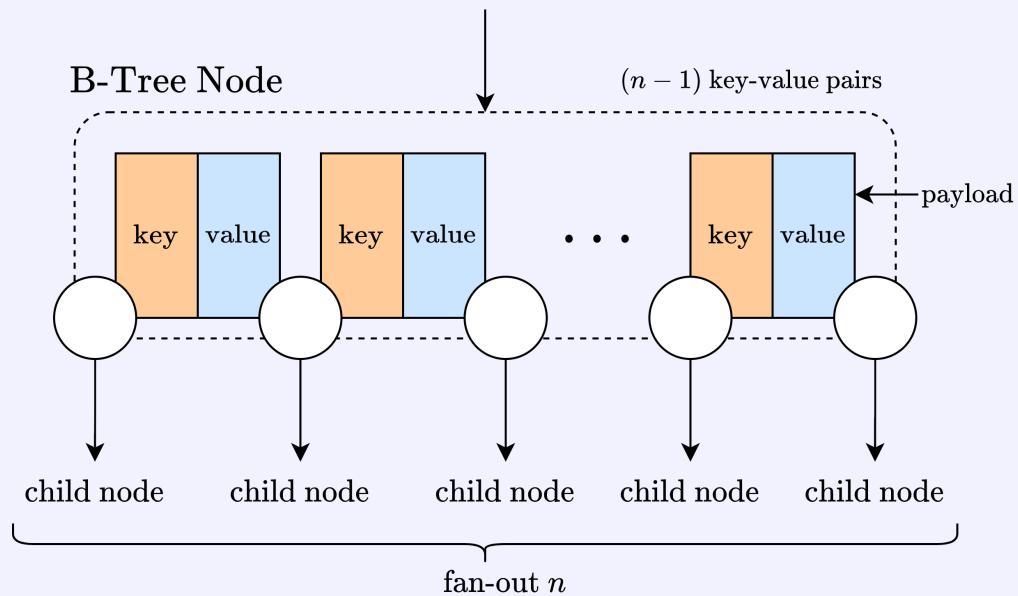


### 4.3.7 B-Trees

Trees are well suited to the requirements of a database:

- Good complexity for equality lookups ( $\log(n)$  tree traversal)
- Easy to update (hash-tables can require a resize and cause a load spike on insert)

Typical balanced tree data structures such as red/black trees, AVL trees are unsuited as they have low fan-out (require a large number of traversals to node spread across many pages → many page faults occur to fetch only a few nodes). Databases are I/O bound (here the I/O is page faults).



A high fan-out, balanced tree.

- Each non-root node must contain at least  $\left\lfloor \frac{n-1}{2} \right\rfloor$  key/value pairs.
- The tree is kept balanced (all leaves at same depth)
- Time complexity for search, insert and delete is  $O(\log n)$

#### 4.3.8 B+ Trees

**UNFINISHED!!!**

##### Maintaining Balance

When a node overflows (full but value needs to be inserted), choose a splitting element and split values one either side into new nodes.

**UNFINISHED!!!**

#### 4.3.9 Foreign Key Indices

Most joins are using a foreign key relation.

- Constraint implies the number of matching tuples is 1 (foreign key  $\rightarrow$  unique primary key)
- A foreign key indices effectively cache/save a join.

# Chapter 5

## Velox

5.1 Motivation

5.2 Overview

5.2.1 Structure

5.3 Use Cases

5.4 Library Components

5.4.1 Data Types

**UNFINISHED!!**

# Chapter 6

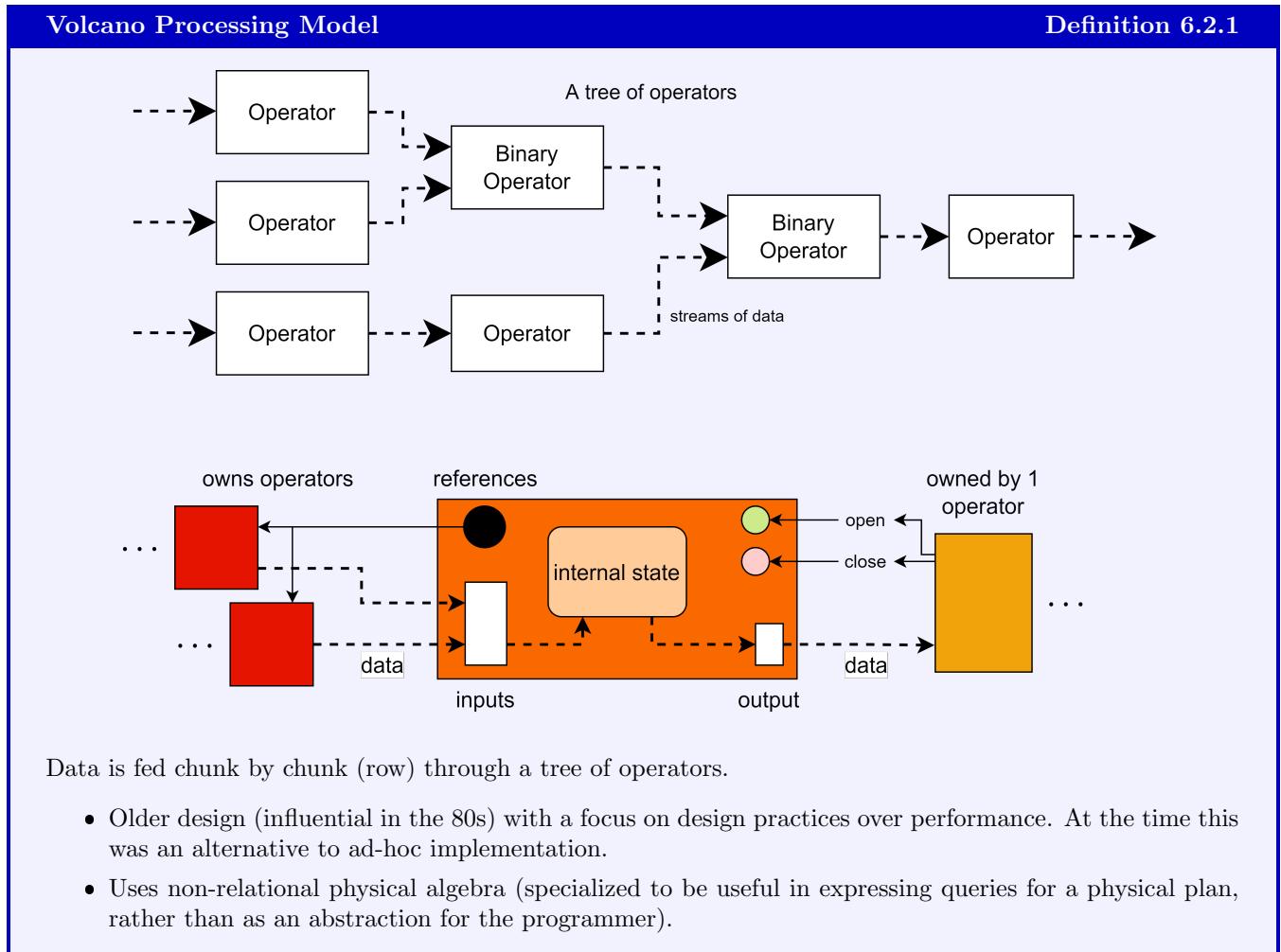
## Processing Models

### 6.1 Motivation

Processing Model	Definition 6.1.1
A mechanism used to connect operators acting on data in a query. <ul style="list-style-type: none"><li>• Choice is critical to database design.</li></ul>	

Function Objects	Definition 6.1.2
References to code that can be passed, invoked, change state and produce values. <pre>#include &lt;functional&gt;  std::function&lt;int(int, int)&gt; add = [ /* captures */ ](int a, int b) { return a + b; }</pre> See C++11 Lambdas <ul style="list-style-type: none"><li>• Can capture variables (value and references to) (also called closures).</li><li>• Used to implement single abstract method classes in some languages (e.g kotlin, java)</li></ul>	

## 6.2 Volcano Processing



**Easy to Implement** Implementation is simple, adding new operators is easy (using operator interface provided). Clean Object-Oriented Design.

**I/O Behaviour** As tuples are consumed as soon as they are produced, no waiting for I/O to create and buffer the next tuple.

**Lots of Calls!** CPU spends much time loading and calling function pointers to operators, predicates and aggregate functions.

### 6.2.1 Operators

A basic interface for operators can be devised as:

```
template <typename T>
struct Operator
{
    virtual void open() = 0;
    virtual void close() = 0;
    virtual optional<T> next() = 0;
};
```

In order to allow the greatest flexibility in using our operators, they are parameterised by `typename T`. In the concrete examples this is set as a *runtime tracked* type `Row` which is variable size, and contains variants of `int`, `char`, `bool`, etc.

We could also swap this out for a reference, or pointer to some *runtime type* to avoid copying.

To keep these examples explicit, an `open()` and `close()` are overriden, rather than using the constructor & destructor.

That said RAII would be useful here:

- Automatically clean up after operators after they are dropped.
- Cannot be used before open/construction, or used after close/destruction.

## Scan

Scans a table already loaded into memory to return its rows.

```
template <typename T>
struct Scan : Operator<T>
{
    using TableType = vector<T>;
    /* Many different operators can have a reference to and read the table.
     * - shared_ptr drops table after it is no longer needed
     * - must avoid copying very large table structure
     */
    Scan(shared_ptr<TableType> t) : _table(t), _index(0) { assert(_table); }

    /* No operation on open / close */
    void open() override {}
    void close() override {}

    optional<T> next() override
    {
        if (_index < (*_table).size())
        {
            return (*_table)[_index++];
        }
        else
        {
            return {};
        }
    }

private:
    shared_ptr<TableType> _table;
    size_t _index;
};
```

## Project

```
template <typename T, typename S>
struct Project : Operator<T>
{
    using Projection = function<T(S)>

    Project(unique_ptr<Operator<S>> child, Projection proj)
        : _child(move(child)), _proj(proj)
    {
        assert(_child);
    }

    void open() override { _child->open(); }
    void close() override { _child->close(); }
```

```

optional<T> next() override
{
    // Note: can be simplified with optional<A>::and_then(function<B(A)>) in C++23
    auto next = _child->next();
    if (next.has_value())
    {
        return _proj(next.value());
    }
    else
    {
        return {};
    }
}

private:
    unique_ptr<Operator<S>> _child;
    Projection _proj;
};

```

## Select

```

template <typename T>
struct Select : Operator<T>
{
    using Predicate = function<bool(T)>

    Select(unique_ptr<Operator<T>> child, Predicate pred)
        : _child(move(child)), _pred(pred)
    {
        assert(_child);
    }

    void open() override { _child->open(); }
    void close() override { _child->close(); }

    optional<T> next() override
    {
        auto candidate = _child->next();
        // keep getting candidates until there are no more, or one is valid.
        while (candidate.has_value() && !_pred(candidate.value()))
        {
            candidate = _child->next();
        }
        return candidate;
    }

private:
    unique_ptr<Operator<T>> _child;
    Predicate _pred;
};

```

## Union

```

template <typename T>
struct Union : Operator<T>
{
    Union(unique_ptr<Operator<T>> leftChild, unique_ptr<Operator<T>> rightChild)
        : _leftChild(move(leftChild)), _rightChild(move(rightChild))
    {
        assert(_leftChild && _rightChild);
    }
}

```

```

}

void open() override
{
    _leftChild->open();
    _rightChild->open();
}
void close() override
{
    _leftChild->close();
    _rightChild->close();
}

optional<T> next() override
{
    auto candidate = _leftChild->next();
    if (candidate.has_value())
    {
        return candidate;
    }
    else
    {
        return _rightChild->next();
    }
}

private:
    unique_ptr<Operator<T>> _leftChild, _rightChild;
};

```

## Difference

### Pipeline Breaker

### Definition 6.2.2

An operator which can only produce its first value/output tuple after all inputs from one or more input operators has been processed.

- Usually requires some kind of buffering (e.g with **Difference**).

Difference breaks the pipeline as we need to know all tuples from one side (the subtracting set) before we can start to produce rows.

```

/* The definition of difference forces the pipeline to be broken (buffering) */
template <typename T>
struct Difference : Operator<T>
{
    Difference(unique_ptr<Operator<T>> fromChild,
                unique_ptr<Operator<T>> subChild)
        : _fromChild(fromChild), _subChild(subChild), _subBuffer()
    {
        assert(_fromChild && _subChild);
    }

    void open() override
    {
        _fromChild->open();
        _subChild->open();

        // buffer all to subtract
        for (auto candidate = _subChild->next(); candidate.has_value();
              candidate = _subChild->next())
    }
}
```

```

    {
        _subBuffer.push_back(candidate);
    }
}
void close() override
{
    _fromChild->close();
    _subChild->close();
}

optional<T> next() override
{
    auto candidate = _fromChild->next();
    // keep getting next until there is no next candidate, or the candidate is
    // not being subtracted
    while (candidate.has_value() && _subBuffer.contains(candidate.value()))
    {
        candidate = _fromChild->next();
    }
    return candidate;
}

private:
    unique_ptr<Operator<T>> _fromChild, _subChild;
    unordered_set<T> _subBuffer;
};

```

## Cartesian/Cross Product

This can be optionally implemented as a *pipeline breaker*.

```

template <typename A, typename B>
struct BreakingCrossProduct : Operator<tuple<A, B>>
{
    BreakingCrossProduct(unique_ptr<Operator<A>> leftChild,
                         unique_ptr<Operator<B>> rightChild)
        : _leftChild(move(leftChild)), _rightChild(move(rightChild)),
        _leftCurrent(), _rightIndex(0), _rightBuffer()
    {
        assert(_leftChild && _rightChild);
    }

    void open() override
    {
        _leftChild->open();
        _rightChild->open();

        // set first left (can be none -> in which case next will never return
        // anything)
        _leftCurrent = _leftChild->next();

        // buffer in the entirety of the right
        for (auto candidate = _rightChild->next(); candidate.has_value();
             candidate = _rightChild->next())
        {
            _rightBuffer.push_back(candidate.value());
        }
    }

    void close() override

```

```

{
    _leftChild->close();
    _rightChild->close();
}

optional<tuple<A, B>> next() override
{
    // invariant: _rightBuffer.size() > _rightIndex >= 0
    if (_leftCurrent.has_value() && !_rightBuffer.empty())
    {
        auto next_val =
            make_tuple(_leftCurrent.value(), _rightBuffer[_rightIndex]);

        _rightIndex++;
        if (_rightIndex == _rightBuffer.size())
        {
            _rightIndex = 0;
            _leftCurrent = _leftChild->next();
        }
    }

    return next_val;
}
else
{
    return {};
}
}

private:
unique_ptr<Operator<A>> _leftChild;
unique_ptr<Operator<B>> _rightChild;
optional<A> _leftCurrent;
size_t _rightIndex;
vector<B> _rightBuffer;
};

```

A Non-pipeline breaking implementation has two phases:

1. Collecting rows from the right child operator, while using the same row from the left.
2. The right child operator has been exhausted, slowly get tuples from the left while traversing tuples collected from the right.

```

template <typename A, typename B>
struct CrossProduct : Operator<tuple<A, B>>
{
    CrossProduct(unique_ptr<Operator<A>> leftChild,
                 unique_ptr<Operator<B>> rightChild)
        : _leftChild(move(leftChild)), _rightChild(move(rightChild)),
          _leftCurrent(), _rightBuffered(), _rightOffset(0)
    {
        assert(_leftChild && _rightChild);
    }

    void open() override
    {
        _leftChild->open();
        _rightChild->open();
        _leftCurrent = _leftChild->next();
    }

    void close() override

```

```

{
    _leftChild->close();
    _rightChild->close();
}

optional<tuple<A, B>> next() override
{
    /* invariants:
     * - _leftCurrent is already set
     * - if there are no more _rightChild to get, then we are iterating over the
     *   _leftChild
     */
    auto rightCandidate = _rightChild->next();
    if (rightCandidate.has_value())
    {
        // still getting content from the right hand side
        _rightBuffered.push_back(rightCandidate.value());
    }
    else if (_rightOffset == _rightBuffered.size())
    {
        // all tuples have been taken from right hand side, now using buffer
        _leftCurrent = _leftChild->next();
        _rightOffset = 0;
    }

    // only return if both sides have values
    if (_leftCurrent.has_value() && !_rightBuffered.empty())
    {
        // get tuple and increment _rightOffset
        return make_tuple(_leftCurrent.value(), _rightBuffered[_rightOffset++]);
    }
    else
    {
        return {};
    }
}

private:
unique_ptr<Operator<A>> _leftChild;
unique_ptr<Operator<B>> _rightChild;
optional<A> _leftCurrent;
vector<B> _rightBuffered;
size_t _rightOffset;
};

```

## Group Aggregation

This is fundamentally a *pipeline breaker*, and must buffer rows prior to `next()`. The algorithm acts in three phases:

1. Buffer tuples from the child.
2. Get the key (column being grouped by e.g `GROUP BY column1`) and aggregation (e.g `SELECT MAX(column2)`) and place in a hashmap.
3. Finally provide rows through `next()`

```

/* We use the template to determine the hash and nextSlot implementations used
 * T      -> type of data provided by the child
 * S      -> data output by the groupBy & aggregation
 * K      -> the type grouped on, produced by a grouping function (K group(T))
 * hash   -> a function to convert a key into a hash
 * nextSlot -> to determine next slot in collisions

```

```

*/
template <typename T, typename S, typename K, size_t nextSlot(size_t),
          size_t hashFun(K), size_t OVERALLOCATE_FACTOR = 2>
struct GroupBy : Operator<S>
{
    using Aggregation = function<S(optional<S>, T)>;
    using Grouping = function<K(T)>;

    GroupBy(unique_ptr<Operator<T>> child,
            Grouping grouping,
            Aggregation aggregation) : _child(move(child)), _grouping(grouping),
                                             _aggregation(aggregation), _hashTable(), _hashTableCursor(0)
    {
        assert(_child);
    }

    void open() override
    {
        _child->open();

        vector<T> childValues;
        for (auto currentValue = _child->next();
             currentValue.has_value();
             currentValue = _child->next())
        {
            childValues.push_back(currentValue.value());
        }

        _hashTable = vector<optional<pair<K, S>>>(childValues.size(), optional<pair<K, S>>());
        for (T val : childValues)
        {
            K key = _grouping(val);
            size_t slot = hashFun(key) % _hashTable.size();
            while (_hashTable[slot].has_value() && _hashTable[slot].value().first != key)
            {
                slot = nextSlot(slot) % _hashTable.size();
            }

            // slot is now correct, either a value present with the same key, or none.
            auto prev_val = _hashTable[slot].has_value() ? _hashTable[slot].value().second : optional<S>();
            _hashTable[slot] = optional<pair<K, S>>(make_pair<K, S>(move(key), _aggregation(prev_val, val)));
        }
    }

    // all values moved into the hashtable, so vector deallocated
}

void close() override
{
    _child->close();
}

optional<S> next() override
{
    while (_hashTableCursor < _hashTable.size())
    {
        auto slot = _hashTable[_hashTableCursor];
        _hashTableCursor++;

        if (slot.has_value())
        {

```

```

        return slot.value().second;
    }
}
return {};
}

private:
Aggregation _aggregation;
Grouping _grouping;
unique_ptr<Operator<T>> _child;
vector<optional<pair<K, S>>> _hashTable;
size_t _hashTableCursor;
};

```

## Operators Composed

We can finally define types to use with our operators.

```

using Value = variant<int, char, bool>;
using Row = vector<Value>;
using Table = vector<Row>;

```

And now build a query from them

```
SELECT table.1, MAX(table.0) FROM table GROUP BY table.1;
```

```

shared_ptr<Table> data = make_shared<Table>(Table{
    {1, 'c', true},
    {1, 'c', false},
    {2, 'c', false},
    {1, 'd', true},
    {3, 'e', false}});

auto scan1 = make_unique<Scan<Row>>(data);
auto scan2 = make_unique<Scan<Row>>(data);
auto cross = make_unique<CrossProduct<Row, Row>>(move(scan1), move(scan2));

Project<Row, tuple<Row, Row>> proj(move(cross), [](tuple<Row, Row> t)
{
    auto vec2 = get<1>(t);
    auto vec1 = get<0>(t);
    vec1.insert(vec1.end(), vec2.begin(), vec2.end());
    return vec1;
});

GroupBy<Row, Row, Value, nextSlotLinear, hashValue>
    groupby(move(scan), groupBySecondCol, aggregateSecondCol);

groupby.open();
for (auto val = groupby.next(); val.has_value(); val = groupby.next())
{
    cout << val.value() << endl;
}
groupby.close();

[ 3 e]
[ 5 c]
[ 1 d]

```

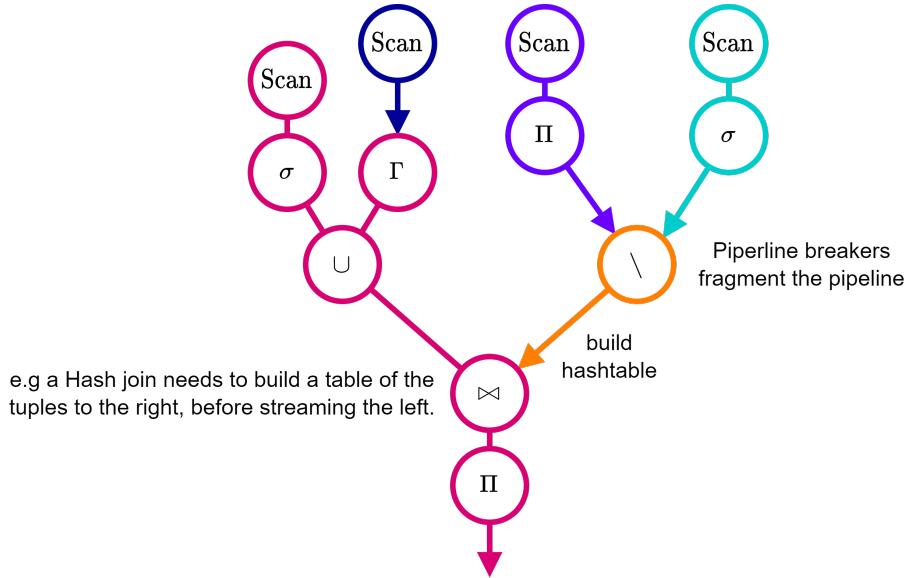
**Run it!**

*Extra Fun! 6.2.2*

The above code is provided with examples in the associated notes repository!

## 6.2.2 Pipelining

### IO Operations



As some operations require buffering, we need to determine how much buffer is required, if this can fit in memory (or disk I/O required), and in which operators.

- If all buffers in a fragment fit in memory, there is no I/O
- Otherwise: sequential access  $\rightarrow$  number of occupied pages, random access/out of order  $\rightarrow$  one page I/O per access (an upper bound & almost certainly an over estimate)

Buffer size depends on the operator, we assume *spanned pages* are used:

- Sorted relations, nested loop buffers  $\rightarrow$  same size as input
- Hashtables have an over-allocation factor (if not known  $\rightarrow$  assume 2)

Finally we assume we know the cardinality of operator inputs and outputs, and the buffer pool size.

Basic GroupBy	Example Question 6.2.1
<pre> CREATE TABLE Customer (     id      i32,     name    STRING, -- Key into compressed dictionary     address STRING, -- Key into compressed dictionary     nation  i32,     phone   i32,     accNum  i32 );  Scan(Customer)     σ<sub>id&gt;250</sub>         Γ<sub>(id),(count)</sub>     </pre>	<p>Customer has 10,000 tuples</p> <p>Strings are represented by a 32 bit integer key into a compressed dictionary.</p> <p><math>\sigma_{id&gt;250}</math> has 30% selectivity.</p> <p><math>\Gamma_{(id),(count)}</math> has grouping cardinality of 9</p> <p>Page size is 64 B</p> <p>Buffer pool is 512 KB</p> <p>1. <i>Scan(Customer):</i></p> $size(Customer) = (6 \times 4) \times 10,000 = 240,000 \Rightarrow pages(Customer) = \left\lceil \frac{240,000}{64} \right\rceil = 3,750$

- Scan reads sequentially, so  $\text{cost}(\text{Scan}(\text{Customer})) = 3,750$  I/O operations.
- Not a pipeline-breaker, so only needs 1 tuple at a time, so no buffer allocation required.

2.  $\sigma_{id < 250}(\dots)$

- Not a pipeline breaker, so no need to buffer.
- No IO costs as child *Scan* operation passes tuple in memory.

3.  $\Gamma_{(id), (count)}$

For the grouping we assume a hashtable overallocation factor of 2, the table contains *count* and grouping attribute (*id*).

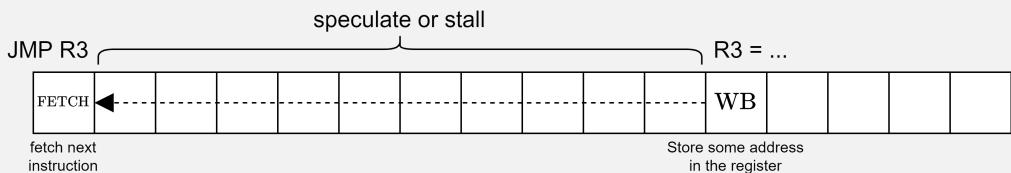
$$\text{size}(\text{GroupBy hashtable}) = 2 \times ((2 \times 4) \times 9) = 144$$

## CPU Efficiency

### Slow Jumps

### Extra Fun! 6.2.3

A jump to a function pointer (e.g a `std::function`, `virtual` method or OUT `(*function_ptr)(A, B, ...)`) is expensive.



- A combined data & control hazard. The address must be known in order to jump, the next instruction after the jump cannot be known until the jump is done.
- There are ways to reduce the stall in hardware (reducing length of pipeline frontend to reduce possible stall cycles, jump target prediction & speculation, delayed jump (allow other work to be done in what would have been stall cycles))
- In software we could load the address to a register many instructions before the jump, and do other useful work between, but often there is little other work to be done.

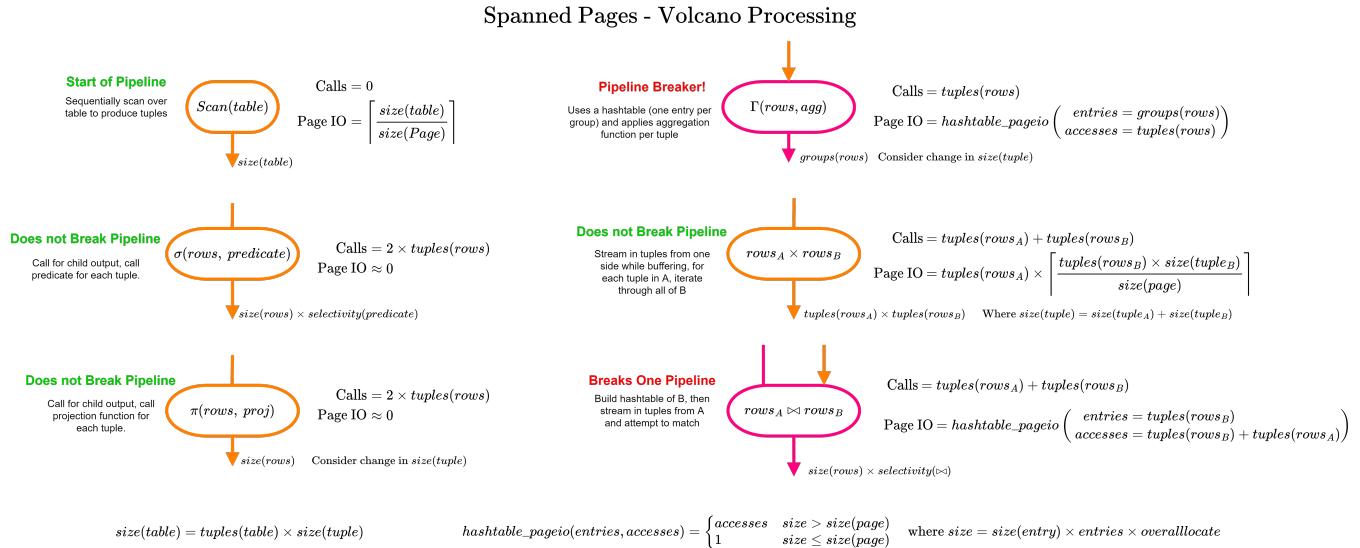
To avoid this cost:

- Jump to an immediate value (typically pc-relative immediate offset in the jump instruction), as the jump location is part of the instruction, there is no hazard. But the function to jump to must be known at compile time. Still affects returns (jump to link register/return address register) (though this should be very fast due to return-address stack branch predictors).
- Inline a function (must be known at compile time)
- Do fewer of these calls to function pointers/virtuals.

For each operation we can count the function calls per tuple

Scan	0	Tuples read straight from buffer.
Select/ $\sigma$	2	Call to read input, call to apply predicate.
Project/II	2	Call to read input, call to projection.
Cross Product (Inner & Outer)	1	Call to read input.
Join	1	Call to read input (comparison and hash can be inlined).
Group-By	2	Read input and call aggregation function.
Output	1	Call to read input and extract to output.

### 6.2.3 Operations Calculations



### Selective

### Example Question 6.2.2

```

CREATE TABLE table (
    a INTEGER,
    b INTEGER,
    c INTEGER
);

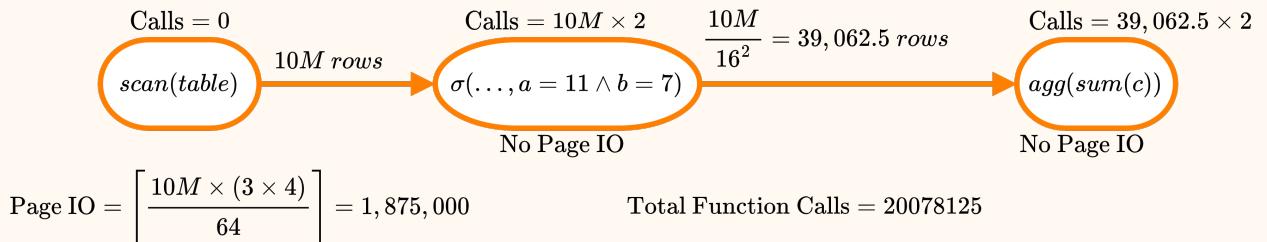
-- note a, b and c are uniform randomly distributed [1-16] inclusive
INSERT INTO table VALUES /* ...10 million rows */ ;

```

-- Evaluate the following query for pageIO and function calls under volcano processing.

```
SELECT sum(c) FROM table WHERE a = 11 AND b = 7;
```

If we assume the WHERE can be done with a single selection, and selection predicate.



## 6.3 Bulk Processing

### Bulk Processing

### Definition 6.3.1

Queries are processed in batches.

- Turn *control dependencies* to *data dependencies* & buffer.
- Pass references to buffers between operators.
- Better locality for code (I-cache) & data.

For example a basic select operator could be implemented on an Nary Table:

- Rather than calling select predicate, provide operators for common predicates (e.g equality)
- Can implement for decomposed storage layout.

```
template <typename V> using Row = vector<V>;
template <typename V> using Table = vector<Row<V>>;

template <typename V>
size_t select_eq(Table<V> &outputBuffer, const Table<V> &inputBuffer, V eq_value, size_t attribOffset) {
    for (const Row<V> &r : inputBuffer) {
        if (r[attribOffset] == eq_value) {
            outputBuffer.push_back(r);
        }
    }
    return outputBuffer.size();
}
```

### Bulking up

### Example Question 6.3.1

Translate the following to use the `select_eq` implementation above.

```
CREATE TABLE Orders (orderId int, status int, urgency int);
```

```
SELECT PendingOrders.* FROM (
    SELECT *
    FROM Orders
    WHERE status = PENDING
) AS PendingOrders
WHERE PendingOrders.urgency = URGENT;
```

```
enum Urgency { URGENT, NOT_URGENT, IGNORE };
enum Status { COMPLETE, IN_PROCESS, PENDING };
```

```
Table<int> Orders{
    {1, COMPLETE, IGNORE},
    {2, PENDING, IN_PROCESS},
    {3, PENDING, URGENT},
    {4, PENDING, URGENT},
};
```

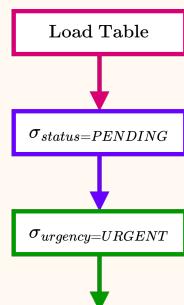
```
Table<int> PendingOrders, UrgentAndPendingOrders;
```

```
select_eq<int>(PendingOrders, Orders, PENDING, 1);
select_eq<int>(UrgentAndPendingOrders, PendingOrders, URGENT, 2);
```

For determining the number of IO operations, bulk operators read all input pages sequentially, and writes to output sequentially.

### Bulk Selection

### Example Question 6.3.2



Compute an estimate of the IO operations for the previous example's query.

- Selectivity of both  $\sigma$  is 25%
- 1,000,000 tuples
- Each tuple contains 3 32 bit integers
- 512 KB cache with 64 B pages

1. Load Page

$$size(Orders) = 1,000,000 \times (3 \times 4) = 12,000,000 \Rightarrow pages(Orders) = \left\lceil \frac{12,000,000}{64} \right\rceil = 187,500$$

Hence 187,500 IO actions

2.  $\sigma_{status=PENDING}$

$$size(PendingOrders) = 250,000 \times (3 \times 4) = 3,000,000$$

$$\Rightarrow pages(PendingOrders) = \left\lceil \frac{3,000,000}{64} \right\rceil = 46,875$$

Hence given the input buffer, there are 46,875 output IO actions.

3.  $\sigma_{urgency=URGENT}$

$$size(PendingAndUrgentOrders) = 62,500 \times (3 \times 4) = 750,000$$

$$\Rightarrow pages(PendingAndUrgentOrders) = \left\lceil \frac{750,000}{64} \right\rceil = \lceil 11,718.75 \rceil = 11,719$$

Hence given the input buffer, there are 11,719 output IO actions.

Hence in total there are  $187,500 + 46,875 + 11,719 = 246,094$  IO actions.

### 6.3.1 By-Reference Bulk Processing

#### By-Reference Bulk Processing

#### Definition 6.3.2

Copying is expensive, so instead of copying rows an identifier/reference is used.

- There is overhead associated with indirection of a reference
- Produced tables can contain many ids out of order & lookups result in random access pattern.

```
// Candidates are indexes into an underlying table
using Candidates = vector<uint32_t>

// To add all rows of a table to some candidates.
template<typename V>
size_t add_candidates(const Table<V>& underlyingBuffer, Candidates& outputRows) {
    for (uint32_t i = 0; i < underlyingBuffer.size(); i++) {
        outputRows.push_back(i);
    }
    return outputRows.size();
}

// An by-reference bulk processing implementation of select
template<typename V>
size_t select_eq(const Table<V>& underlyingBuffer, Candidates& outputRows,
                 const Candidates& inputRows, V eq_value, size_t attribOffset) {
    for (const uint32_t index : inputRows) {
        if (underlyingBuffer[index][attribOffset] == eq_value) {
            outputRows.push_back(index);
        }
    }
    return outputRows.size();
}
```

We can then demonstrate the previous example with the following query

```

Candidates OrdersCandidates, PendingOrders, UrgentAndPendingOrders;
add_candidates(Orders, OrdersCandidates);
select_eq<int>(Orders, PendingOrders, OrdersCandidates, PENDING, 1);
select_eq<int>(Orders, UrgentAndPendingOrders, PendingOrders, URGENT, 2);

```

## Page Access Probability

When estimating page IO we must consider access to candidates:

- Access to candidate vectors can result in page IO.
- Indexes from candidate vectors are ordered, but may be spread across the underlying table's pages.

Probability of a page being touched, given  $s$  selectivity of tuples and  $n$  tuples per page.

$$p(s, n) = 1 - \underbrace{(1-s)^n}_{\text{no tuples accessed}}$$

Hence for a selection:

$$\text{PageFault} = p(s, n) \times \text{pages}(\text{underlying}) \text{ where } \begin{cases} s &= \text{selection selectivity} \\ n &= \frac{\text{page size}}{\text{tuple size}} \end{cases}$$

### 6.3.2 Decomposed Bulk Processing

Decomposed storage was introduced as a consequence of bulk processing:

- By storing columns contiguously, page faults are reduced by accessing a column.
- Reduces pressure on space occupied by underlying table in buffer pool/cache (only need relevant columns loaded).

## IO Operations

We must adapt the scheme used for by-reference bulk processing to account of decomposed storage.

- Only need to consider the size of the data in the column being accessed.

Bulk Columns	Example Question 6.3.3
<b>UNFINISHED!!!</b>	

# Chapter 7

# Optimisation

## 7.1 Motivation

*"Users expect miracles! . . . Data management systems can actually accommodate some . . . - Holger Pirk"*

- Users want zero-overhead, the system should be as fast as hand-written & optimised code.
- The database is expected to learn from data (e.g second run of a query is faster)
- System must be highly flexible (users can create relations, indices, build complex queries without needing to upgrade/reconfigure/recompile any part of the DBMS)

In reality current *DBMS* generally succeed in meeting these *miraculous* expectations.

### 7.1.1 Query Optimisers vs Optimising Compilers

A query optimiser is similar to a compiler's optimiser:

- Representation of code is transformed through several representations, some logical (e.g AST, three address code), some physical (e.g x86 specific IR, assembly representation)
- Correctness under optimisations (**primary objective**), performance of optimiser queries (**secondary objective**).
- Limitations on time to optimise (i.e developers don't want to wait excessively long to compile simple programs)

The main difference is timing of access to code and input data.

	Code/Query	Input Data
Compiler Optimiser	At compile time	Unknown
Query Optimiser	At query time	Known before query

#### Profile Guided Optimisation

#### Extra Fun! 7.1.1

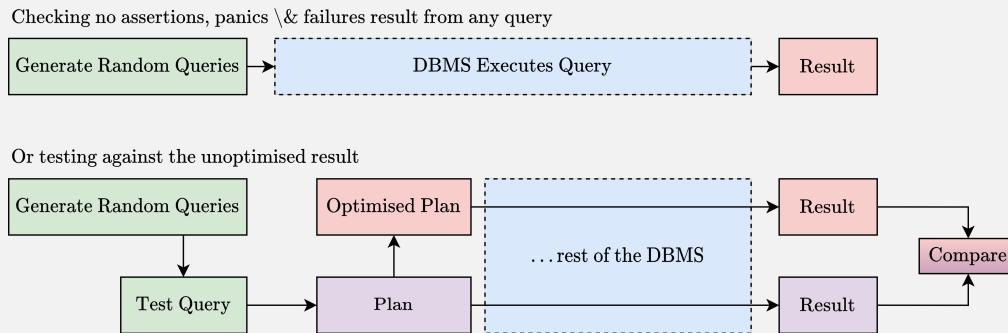
A compiler optimiser (at compile) does not have access to the input data (at runtime). However this is not entirely *technically* true. We can compile an instrumented version of the code, run with some representative input data, profile and provide this feedback to the compiler to guide optimisation.

```
g++ -fprofile-generate myprog.cpp # Compile instrumented version
./myprog.cpp                      # Generates myprog.gcda
g++ -fprofile-use myprog.cpp        # Use profile when optimising
```

Correctness is difficult.

- ANSI SQL semantics are not formally defined (though some have been developed).
- Need to test against complex queries, numerous edge cases, with many combinations of optimisations (much the same as with compiler's optimisers).

One common practice for testing compilers (and DBMS) is to randomly generate potential queries, and then test for differences in results from optimised and un-optimised.



For example SQLsmith can be used to generate random SQL queries, and has been used to test and find bugs in Postgres, sqlite3, monetdb and more (see the score list).

## 7.1.2 Query Equivalence

### Semantic Equivalence

### Definition 7.1.1

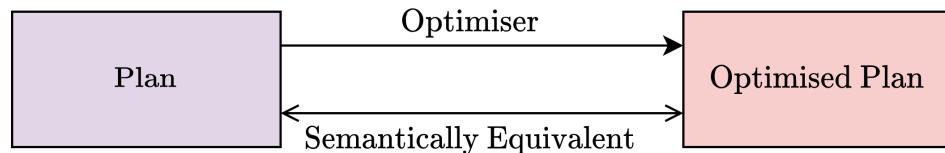
Plans are *semantically equivalent* if they provably produce the same output on any dataset.

### Closure (Mathematics)

### Definition 7.1.2

(*Simplified*) A set is closed under an operation if the operation produces elements of the same set.

- $\mathbb{N}$  is closed under  $+$ , but not under  $-$  (can produce negative numbers)
- *Relational algebra* is closed (the set of possible relations is closed under the operators of the algebra).



As *relational algebra* is closed, operators are easily composable.

- We can determine equivalences between compositions of operators.
- Substitutions of a part of a plan with an equivalent, results in a new equivalent plan.
- We can use this to transform plans into more optimal (but equivalent) plans.

### MonetDB Optimiser

### Extra Fun! 7.1.3

MonetDB is an open source, in-memory, decomposed database. Its optimiser includes implementations for the optimisations discussed in this chapter (e.g selection pushdown)

## 7.2 Peephole Transformations

*"An equivalent transformation of a subplan is an equivalent transformation of the entire plan."*

A set of rules for transforming small subplans (peephole) is applied while traversing the plan.

This is the same idea as peephole optimisations discussed in the WACC project and 50006 - Compilers.

`mov r1, r1 ; redundant move`

`str r4, [sp, #8] ; overwritten store`

`str 13, [sp, #8]`

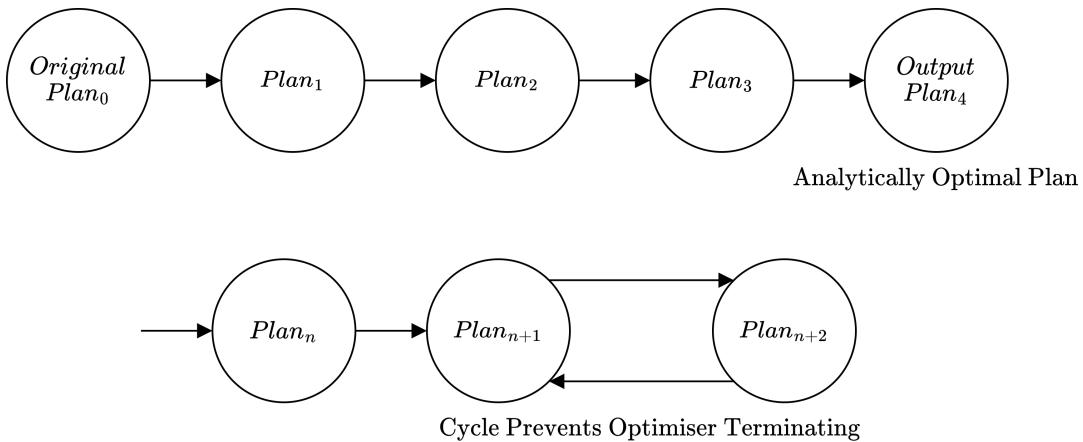
- Need some order with which to traverse the plan
- Need a set of patterns/rules to apply.

### 7.2.1 Avoiding Cycles

#### Analytically Optimal Plan

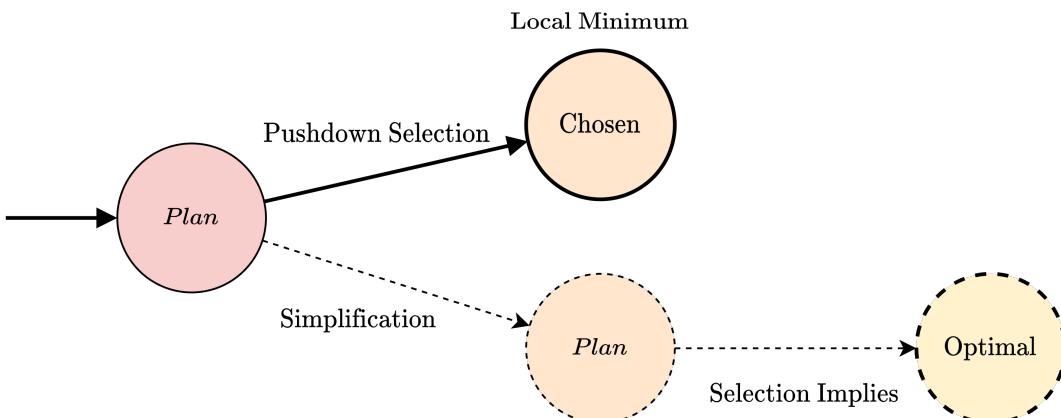
#### Definition 7.2.1

The final plan output of the optimiser (not necessarily the most optimal plan).



Avoiding this requires careful rule selection.

### 7.2.2 Branches



As many possible rules may be applied, some strategy is needed to determine which to apply (e.g just order the rules).

#### Simplicity

Very easy to implement (particularly with pattern matching).

#### Time

Matching and applying rules is faster than more holistic approaches.

#### Verifiability

Can check each rule for correctness by checking if all rules produce semantically equivalent sub-plans.

#### Composability

Can easily add new rules to be composed with previous, rules can enable new rules to be applied.

<b>Loops</b>	Developer must be careful to not introduce potential loops in rule application.
<b>Local Optima</b>	Typically many choices of rule to apply $\Rightarrow$ local optima.

## 7.3 Classifying Optimisation

**Algorithm** The implementation of operators (e.g joins).

**Data** Data & metadata held by the system (e.g cardinalities, histograms)

		Algorithm	
		Agnostic	Aware
		Logical	Physical
<b>Data</b>	Agnostic	Rule-Based	•
	Aware	Cost-Based	•

In DBMS optimisations are defined as operating on *logical* or *physical plans*, and are either *rule-based* or *cost-based*.

<b>Logical</b>	<i>Algorithm-Agnostic</i>	Deals only with relational algebra.
<b>Physical</b>	<i>Algorithm-Aware</i>	Can use different operator implementations, indices etc.
<b>Rule-Based</b>	<i>Data-Agnostic</i>	Applying optimisation rules that are almost always beneficial.
<b>Cost-Based</b>	<i>Data-Aware</i>	Using data to estimate the cost of operations in order to determine which transformations to apply (e.g reordering selections based on each's estimated selectivity).

## 7.4 Logical Optimisation

In order to demonstrate logical optimisation we use a representation of (pseudo) relational algebra in Haskell.

```
data Operator =
  Scan Table
  | Select      Operator Predicate
  | Project     Operator RowTrans
  | Product     Operator Operator
  | Join        Operator Operator Predicate
  | Difference   Operator Operator
  | Union        Operator Operator
  | Aggregation Operator AggFun
  | TopN         Operator SortBy
```

- Purely logical representation, Processing model & operator implementations not specified.
- Other functions for predicting cost, ordering predicates defined
- Using `data` to allow for easy pattern matching, rather than using an operator typeclass.

We include basic functions for applying transformations to the plan:

```
-- Apply some transformation to all children of an operator
apply :: (Operator -> Operator) -> Operator -> Operator

-- Maybe peephole optimise operator (do not traverse to children)
type Peephole = Operator -> Maybe Operator

-- Optimise a plan
type Optimiser = Operator -> Operator
```

hence we can create functions to take a set of *rules* and some *traversal* and create an optimiser we can apply to plans. For example:

```
-- Continue traversing until making an optimisation, then return to root.
-- As optimisations on either side of a join, difference, or union are
-- independent, traverse both independently (with apply).
root :: Peephole -> Optimiser
root peep orig
= case peep orig of
  Just opt -> opt
  Nothing -> apply (root peep) orig
```

All that remains is to determine the `Peephole`'s rules.

One way to further simplify the representation is to embed RA as a DSL within another language. Racket (*the language oriented programming language*) is designed for this. Have a go with your own implementation!

### 7.4.1 Rule Based Logical Optimisation

The optimiser has a set of (almost) universally beneficial rules applied to transform the plan.

Some basic assumptions from which to derive rules include:

- Higher cardinality (more tuples)  $\Rightarrow$  Higher Cost
- Joins usually increase cardinality, or leave unchanged
- Selections reduce cardinality
- Aggregations reduce cardinality
- Data access is more expensive than function evaluation (can assume generally, without exposing operator implementation)

**Portable** Implementation independent (i.e can change processing model without needing to change optimisations - reduced developer maintenance requirements).

**Robust** Small changes in the data or algorithm do not dramatically change performance

**Wrong** The transformations need to be almost always beneficial, so must be conservative with choosing rules. A wrong rule can significantly reduce performance.

**Brittle** A rule removal/addition can result in significant performance changes.

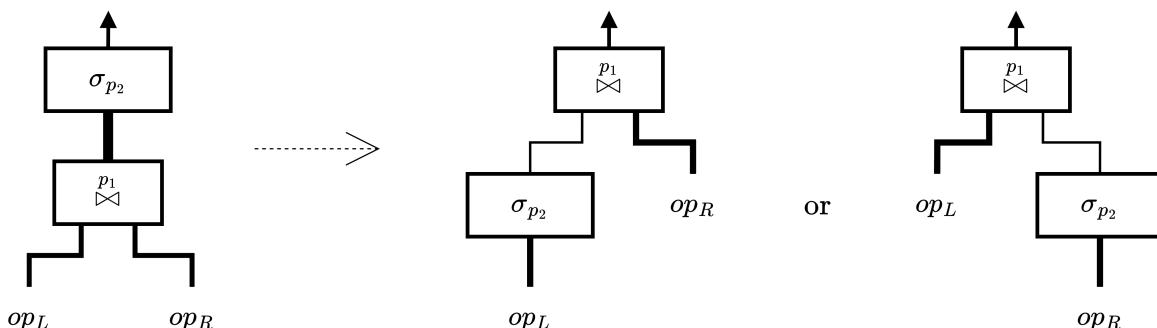
**Unprincipled** Rules tend to be ad-hoc or arbitrary, they are backed by assumptions - not information about workload.

**Loops** As with *peephole* in general.

```
-- creating peephole opt for logical rule-based optimisation
logicalRuleBased :: Peephole
```

```
-- at the end is a catch-all base case
logicalRuleBased _ = Nothing
```

#### Selection Pushdown



Selections can be *pushed down* through joins if they only use attributes from one side of the join.

- As selections are pipelineable, this often a good optimisation when the underlying processing model is volcano.

`SELECT * FROM opL JOIN opR WHERE p2;`

... is optimised to ...

```

SELECT * FROM (SELECT * FROM opL WHERE p2) JOIN opR;
-- or
SELECT * FROM opL JOIN (SELECT * FROM opR WHERE p2);
-- assuming attributes names of opR and opL different
logicalRuleBased (Select (Join opL opR p1) p2)
| attributes opR `containsAll` selectCols p2 = Just (Join opL (Select opR p2 s2) p1 s1)
| attributes opL `containsAll` selectCols p2 = Just (Join (Select opL p2 s2) opR p1 s1)

```

Dont push me down!

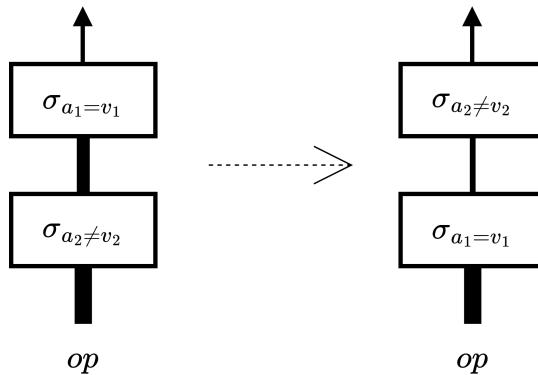
Example Question 7.4.1

Is selection pushdown ever not very beneficial, provide some edge cases?

- If the selectivity of the selection is 100% and the join does not increase cardinality (no benefit).
- If the join significantly reduces cardinality.

**UNFINISHED!!!**

### Selection Ordering



Reordering selections to reduce cardinality at the earliest possible operator.

- We infer which selection has the lowest selectivity using a heuristic
- A common heuristic for comparison operators: == < ( < and > ) < ( <= and >= ) < <>.

SELECT \* FROM ( SELECT \* FROM op WHERE a2 <> v2 ) WHERE a1 == v1;

... is optimised to ...

SELECT \* FROM ( SELECT \* FROM op WHERE a1 == v1 ) WHERE a2 <> v2;

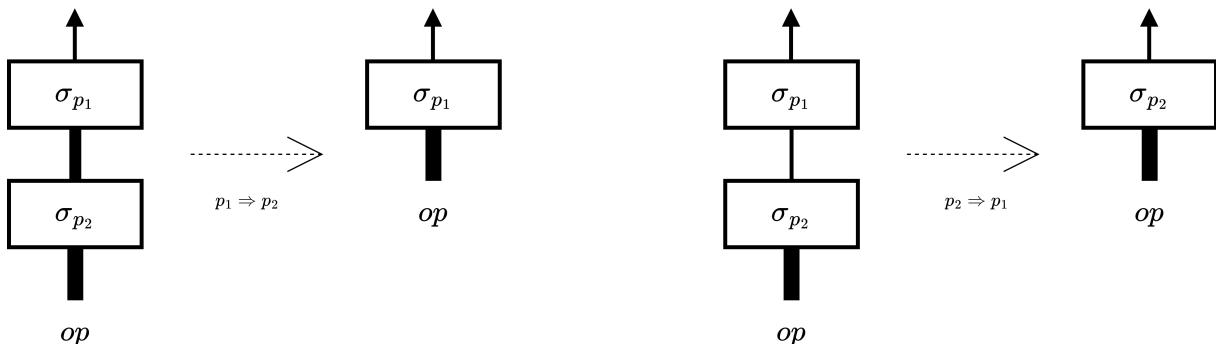
logicalRuleBased

```

(Select (Select op p1) p2) | p2 `predicateLess` p1 -- (with EQ < NEQ)
= Just (Select (Select op p2) p1)

```

### Implication



Given one selection implies the other, we can eliminate another.

```
SELECT * FROM (SELECT * FROM op WHERE a1 == v1) WHERE a1 == v1 AND a2 == v2
```

... is optimised to ...

```
SELECT * FROM op WHERE a1 == v1 AND a2 == v2
```

```
logicalRuleBased
(Select (Select op p1) p2)
| p1 `predicateImplies` p2 = Just (Select op p1)
| p2 `predicateImplies` p1 = Just (Select op p2)
```

More sophisticated rules for simplifying, combining and eliminating selections are possible.

## 7.4.2 Cost Based Logical Optimisation

A cost metric is defined to determine which optimised plans are *better/worse*.

```
-- Types for query optimisation
type Selectivity = Double
type Cost        = Double

-- a function to determine the selectivity of a predicate
selectivity :: Predicate -> Cost

-- a heuristic for cost, using an estimate for the number of tuples output by an operator
sizeCost :: Operator -> Cost
sizeCost op = case op of
  Scan      t      -> fromIntegral (tableSize t)
  Select    op    p   -> selectivity p * sizeCost op
  Project   op    _   -> sizeCost op
  Product   opL opR  -> sizeCost opL * sizeCost opR
  Join      opL opR p -> selectivity p * sizeCost opL * sizeCost opR
  Difference opL opR  -> max (sizeCost opL) (sizeCost opR)
  Union     opL opR  -> sizeCost opL + sizeCost opR
  Aggregation _ af   -> aggGroups af
  TopN      op    _ n  -> min (sizeCost op) n
```

**Selectivity** needs to get an estimate. We will consider the basic case of an equality selection  $\sigma_{a=v}$  where the possible values of  $v$  are for attribute  $a$  are known.

### Uniform Distribution

If we assume all values are equally likely:

$$\text{selectivity}(a = v) \triangleq \frac{1}{\text{number of distinct values}}$$

### Histograms

Store the frequency of values in a table.

	$histogram_a$					
values	$v_1$	$v_2$	$v_3$	$\dots$	$v_n$	
frequency	$c_1$	$c_2$	$c_3$	$\dots$	$c_n$	
					$\text{selectivity}(a = v) \triangleq P(a = v) \equiv \frac{histogram_a.v}{histogram_a.total}$	

- Must retain and update a histogram for each attribute, with a count for each unique value.
- Histograms can be binned (like bitmap indices) when the number of unique values is large.

when evaluating multiple equalities, we assume *attribute independence*, and hence:

$$\text{selectivity}(a_1 = v_1 \wedge \dots \wedge a_n = v_n) \equiv P(a_1 = v_1) \times \dots \times P(a_n = v_n) = \frac{histogram_{a_1}.v_1}{histogram_{a_1}.total} \times \dots \times \frac{histogram_{a_n}.v_n}{histogram_{a_n}.total}$$

SparkSQL's catalyst optimiser uses binned histograms as implemented here

## Multidimensional Histograms

Often attribute values are correlated (e.g largest orders tend to be urgent).

		$histogram_{(a_1, a_2)}$				
		attribute $a_1$				
		$v_{a_1 1}$	$v_{a_1 2}$	$v_{a_1 3}$	$\dots$	$v_{a_1 n}$
		$v_{a_2 1}$	$c_{(1,1)}$	$c_{(2,1)}$	$c_{(3,1)}$	$c_{(4,1)}$
		$v_{a_2 2}$	$c_{(1,2)}$	$c_{(2,2)}$	$c_{(3,2)}$	$c_{(4,2)}$
attribute $a_2$		$v_{a_2 3}$	$c_{(1,3)}$	$c_{(2,3)}$	$c_{(3,3)}$	$c_{(4,3)}$
		$\vdots$	$c_{(1,4)}$	$c_{(2,4)}$	$c_{(3,4)}$	$c_{(4,4)}$
		$v_{a_2 n}$	$c_{(1,5)}$	$c_{(2,5)}$	$c_{(3,5)}$	$c_{(4,5)}$
						$c_{(5,5)}$

- Store multiple histograms to show frequencies of attribute values, given other attribute's value.
- Number of histograms grows combinatorially with number of tables. **NEEDS IMPROVEMENT!!!**
- Reducing the number of histograms, but still producing good selectivity estimates is an open area of research.

$$selectivity(a_1 = v_1 \wedge a_2 = v_2) = P(a_1 = v_1 | a_2 = v_2) \times P(a_2 = v_2) = \frac{histogram_{(a_1, a_2)}.(v_1, v_2)}{histogram_{(a_1, a_2)}.total}$$

## 7.5 Physical Optimisation

### Physical Plan

### Definition 7.5.1

A plan containing implementation specific information, and describing how the query should be physically executed.

- Operator implementations (e.g which join: sort-merge, hash, nested loop, index based join etc)
- Costs of different implementations (e.g hash join vs nested-loop → time versus memory)
- Available indices & data structure choices (e.g type of hashmap, hash function)

Physical plan optimisation focuses on optimising the plan for the specific system the query is executed on.

The cost metric different types of cost (e.g time versus memory)

- Produced tuples
- Page faults
- Intermediate buffer sizes
- (Volcano Processing) function calls
- Storage access & availability

We can then decide if a rule is universally beneficial (for *rule-based*), or determine which possible plan is lowest cost (*cost-based*)

### 7.5.1 Rule Based Physical Optimisation

Much like *logical rule-based optimisation*, (almost) universally beneficial (given the decided cost metric) rules to improve performance.

Data structures	Always use hash map with rehashing for probe if expected collisions are high.
Parallelism	always use parallel sort for <code>ORDER BY</code> , always partition hash joins
Using Indices	If foreign key index exists, always use for foreign key join, if getting range always use available bitmap or B+ tree index.
Cache	Always use cache-conscious partitioning to improve locality.

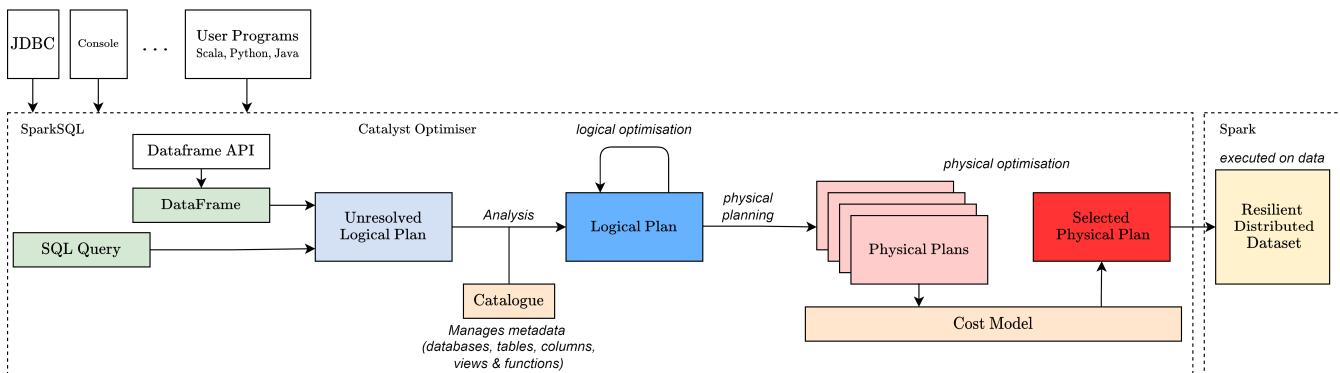
## 7.5.2 Cost Based Physical Optimisation

Much like *logical cost-based optimisation* but on a physical plan (using implementation specific details).

Data	Consider cardinalities & how this affect operator choice (e.g choose sort-merge join over hash if the required hashtable is too large for the buffer pool).
Hardware	Function call overhead (for this architecture), buffer pool size, access latencies, available parallelism (hardware threads).
Algorithm	Must consider how algorithms expected costs change with parameters (e.g cardinality)

This is the current state of the art in optimisation.

## 7.6 SparkSQL



- Rule-based logical optimiser and cost-based physical optimiser
- Rather than reapplying the physical plan optimiser repeatedly on one plan, multiple possible candidate plans are produced and evaluated (negates local optima problem at the cost of generating many physical plans).

Logical rules are expressed as extensions of a `Rule[LogicalPlan]` interface. For example expression simplification and constant folding can be found in the expression optimiser.

```

/**
 * Simplifies boolean expressions:
 * 1. Simplifies expressions whose answer can be determined without evaluating both sides.
 * 2. Eliminates / extracts common factors.
 * 3. Merge same expressions
 * 4. Removes `Not` operator.
 */
object BooleanSimplification extends Rule[LogicalPlan] with PredicateHelper {
  def apply(plan: LogicalPlan): LogicalPlan = plan.transformWithPruning(
    _.containsAnyPattern(AND, OR, NOT), ruleId) {
    case q: LogicalPlan => q.transformExpressionsUpWithPruning(
      _.containsAnyPattern(AND, OR, NOT), ruleId) {
        case TrueLiteral And e => e
        case e And TrueLiteral => e
        case FalseLiteral Or e => e
        case e Or FalseLiteral => e
        // ...
      }
    // ...
  }
}
  
```

# Chapter 8

## Transactions

### 40007 - Introduction to Databases

### Extra Fun! 8.0.1

Histories, anomalies and basic concurrency control are also taught in the 40007 - Introduction to Databases module.

### 8.1 SQL Transaction

```
BEGIN TRANSACTION T1  
  
-- commands to be run, for example:  
SELECT * FROM Orders;  
INSERT INTO Customers VALUES ("bob", 2, 44);  
  
END TRANSACTION -- transaction is committed or aborted
```

#### Transaction

#### Definition 8.1.1

A block of sql statements that can be run on a database, transactions respect the *ACID properties*.

Many transactions can be executed on a database concurrently, we can reason about a *serialization graph*:

- Shows which transactions observe the effects of other transactions.
- Cannot have cycles → if a DBMS observes a cycle will occur, it must recover (e.g by aborting a transaction)

#### Graph cycles

#### Example Question 8.1.1

Is a cycle present in the serialization graph from the following transactions?

```
BEGIN TRANSACTION T1  
INSERT INTRO table1 VALUES (1,9);  
SELECT sum(column1) FROM table1;  
END TRANSACTION
```

```
BEGIN TRANSACTION T2  
INSERT INTRO table1 VALUES (17,90);  
SELECT sum(column1) FROM table1;  
END TRANSACTION
```

Yes as **TRANSACTION T1** reads from **TRANSACTION T2's** insertion (17, 90) and vice versa for insertion (1, 9).

### 8.1.1 ACID Properties

#### Atomicity

#### Definition 8.1.2

Transactions are completed in entirety (committed), or not at all (aborted).

## Consistency

## Definition 8.1.3

Transactions bring the database between states where explicit and implicit constraints are satisfied & the database is valid. There can be inconsistency between states/within a transaction.

## Isolation / Serializability

## Definition 8.1.4

The observable state of a database after all transactions are committed must be equivalent to some serial execution.

- Can create a *serialization graph* with no cycles.

## Durability / Recoverability

## Definition 8.1.5

A committed transaction does not depend on the effect of an uncommitted transaction. The results of committed transactions are persistent.

- Hence it is safe to abort any uncommitted transaction.
- Once committed, the results of a transaction are durable to failure (e.g power failure).

## 8.2 Histories

### Read/Write Locks

### Definition 8.2.1

**Write/Exclusive** Only lock holder can hold write lock for object  $o_1$ .

**Read/Shared** Any number of other read locks on  $o_1$  can be held.

- Many different locking schemes can be implemented → impact possible anomalies and performance.
- Can lock different object types for differing levels of granularity (an entire database, a table, a set of tuples, a single tuple or even a single value in a tuple).

Read/Write locks exist in many languages (e.g `std::shared_mutex` in C++).

		Transaction 1	
		Read	Write
Transaction 2	Read	No Conflict	Conflict!
	Write	Conflict!	Conflict!

We can formalise *transactions* by their read/write operations, and by the locks they acquire to perform these.

$rl_1[o_1]$  Transaction 1 acquires a read lock on object  $o_1$ .

$ru_1[o_1]$  Transaction 1 releases a read lock on object  $o_1$ .

$r_1[o_1]$  Transaction 1 reads from object  $o_1$ .

$wl_2[o_3]$  Transaction 2 acquires a write lock on object  $o_3$ .

$wu_2[o_3]$  Transaction 2 releases a write lock on object  $o_3$ .

$w_2[o_3]$  Transaction 2 writes to object  $o_3$ .

$c_7$  Transaction 7 commits.

$a_5$  Transaction 5 aborts.

$e_1$  Transaction 1 commits or aborts.

We can order operations using *first*  $\prec$  *second*.

## 8.3 Anomalies

### Dirty Read / Read After Write / Uncommitted Dependency

#### Definition 8.3.1

A transaction reads uncommitted data.

$$w_1[o] \prec r_2[o] \prec e_1$$

```
BEGIN TRANSACTION T1
```

```
SELECT a FROM table WHERE b = 1;
```

```
END TRANSACTION
```

-- committed T1 depends on uncommitted T2

```
BEGIN TRANSACTION T2
```

```
UPDATE table SET a = 5 WHERE b = 1;
```

```
END TRANSACTION
```

### Non-Repeatable Read

#### Definition 8.3.2

Reads within the same transaction, of the same rows, contain different values.

```
BEGIN TRANSACTION T1
SELECT * from table;
```

```
BEGIN TRANSACTION T2
```

```
UPDATE table SET a = 9 WHERE b = 3;
END TRANSACTION
```

```
SELECT * from table;
END TRANSACTION
```

### Phantom Read

#### Definition 8.3.3

Reads within the same transaction return a different set of rows. Hence some rows were *phantom*.

- For example two identical selects producing different results imply some *phantom* data has been read.

```
BEGIN TRANSACTION T1
SELECT * from table;
```

```
BEGIN TRANSACTION T2
```

```
DELETE FROM table; -- delete all rows
END TRANSACTION
```

```
SELECT * from table;
END TRANSACTION
```

### Dirty Write / Write After Write

#### Definition 8.3.4

A transaction overwrites uncommitted data from another transaction.

$$w_1[o] \prec w_2[o] \prec e_1$$

```
BEGIN TRANSACTION T1
```

```
UPDATE table SET a = 9 WHERE b = 1;
UPDATE table SET a = 9 WHERE b = 3;
```

```
END TRANSACTION
```

```
BEGIN TRANSACTION T2
```

```
UPDATE table SET a = 5 WHERE b = 1;
```

```
UPDATE table SET a = 5 WHERE b = 3;
END TRANSACTION
```

- If **TRANSACTION T1** overwrites uncommitted updates from **TRANSACTION T2**, then we will get a mixture of **a = 9 OR a = 5**.
- Under these circumstances there is no equivalent serial execution

## Write Skew

## Definition 8.3.5

Concurrent transactions read an overlapping range of rows and commit disjoint updates without seeing the other's update.

```
BEGIN TRANSACTION T1  
-- read a  
  
UPDATE table SET c = a WHERE b = 1;  
END TRANSACTION  
-- a <> c (a and b has been swapped)
```

```
BEGIN TRANSACTION T2  
-- read c  
UPDATE table SET a = c WHERE b = 1;  
END TRANSACTION
```

## Inconsistent Analysis

## Definition 8.3.6

A transaction reads an *inconsistent* view of the database state.

$$r_1[o_a] \prec w_2[o_a], w_2[o_b] \prec r_1[o_b]$$

**UNFINISHED!!!**

```
BEGIN TRANSACTION T1  
  
SELECT sum(a) FROM table;  
  
SELECT sum(a) FROM table;  
-- sum reads some a = 9 and some a = 17  
END TRANSACTION
```

```
BEGIN TRANSACTION T2  
UPDATE table SET a = 9 WHERE b = 1;  
  
UPDATE table SET a = 17 WHERE b = 3;  
END TRANSACTION
```

## Lost Update

## Definition 8.3.7

A write from a transaction is overwritten by another update using outdated information.

$$r_1[o] \prec w_2[o] \prec w_1[o]$$

```
BEGIN TRANSACTION T1  
WITH old AS (SELECT a FROM table WHERE b = 1)  
  
UPDATE table SET a = (  
    SELECT a + 4 FROM OLD  
) WHERE b = 1;  
END TRANSACTION
```

```
BEGIN TRANSACTION T2  
UPDATE table SET a = 9 WHERE b = 1;  
  
END TRANSACTION
```

## 8.4 Isolation Levels

### Read Uncommitted

### Definition 8.4.1

Dirty Write	Dirty Read	Write Skew	Inconsistent Analysis	Lost Update	Ph-Read	Non-Rep Read
Prevented	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed

Reads occur immediately (can read uncommitted data), writers wait for other writers to commit (serial order for writers).

- All readonly queries can execute immediately and in parallel.
- Susceptibility to anomalies means it is rarely the default isolation level.

Read Committed							Definition 8.4.2
Dirty Write Prevented	Dirty Read Prevented	Write Skew Allowed	Inconsistent Analysis Allowed	Lost Update Allowed	Ph-Read Allowed	Non-Rep Read Allowed	
Readers wait for writers to commit. All writes of a transaction are atomically made available at commit time.							
<ul style="list-style-type: none"> <li>Each row is read/write locked, read locks acquired &amp; dropped as needed, write locks held until commit/abort.</li> </ul>							

Repeatable Read							Definition 8.4.3
Dirty Write Prevented	Dirty Read Prevented	Write Skew Prevented	Inconsistent Analysis Prevented	Lost Update Prevented	Ph-Read Allowed	Non-Rep Read Prevented	
Strengthen's read committed to guarantee any repeated read in a transaction returns the same result.							
<ul style="list-style-type: none"> <li>Phantom reads can occur (different rows → different reads)</li> <li>Each row is read/write locked, both read &amp; write locks held until commit/abort.</li> </ul>							

Serializable							Definition 8.4.4
Dirty Write Prevented	Dirty Read Prevented	Write Skew Prevented	Inconsistent Analysis Prevented	Lost Update Prevented	Ph-Read Prevented	Non-Rep Read Prevented	
Readers get full isolation, execution is serializable.							
<ul style="list-style-type: none"> <li>Restrictive &amp; expensive → never the default isolation level.</li> <li>Each range of rows (e.g table) affected by a transaction is read/write locked for the entire transaction.</li> </ul>							

## 8.5 Concurrency Schemes

- Serializability is required → Use 2PL  
 Conflicts expected to be Low → Use OCC lowest overhead  
 Conflicts expected to be high → Use MVCC

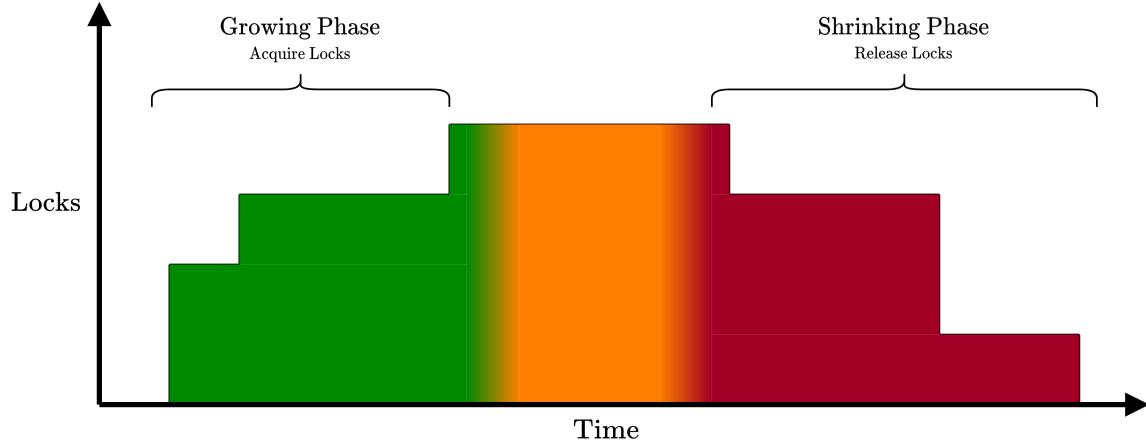
### 8.5.1 Serial Execution

No concurrency, execute all transactions in sequential order.

<b>No Anomalies</b>	Solves all aforementioned anomalies.
<b>Simple Implementation</b>	Easy to implement: No concurrency ⇒ No problems!
<b>Latency</b>	As transactions cannot occur concurrently, they must be queued and the user must wait. Very poor performance under load.
<b>Underutilisation</b>	Limited usage of hardware → limit to how much individual queries can be parallelised to use cores, more transactions in parallel solves this.
<b>Not Scalable</b>	Cannot apply to larger databases (e.g supporting millions of large queries per second).

A better approach is to take a practical approach to concurrency (limit if necessary for correctness, otherwise maximise), and to accept some anomalies (allow the user to configure which are acceptable).

### 8.5.2 Two-Phase Locking (2PL)



- Transaction acquires locks required in *growth phase*, and releases in *shrinking phase*.
- Acquires locks on objects before reading/writing.

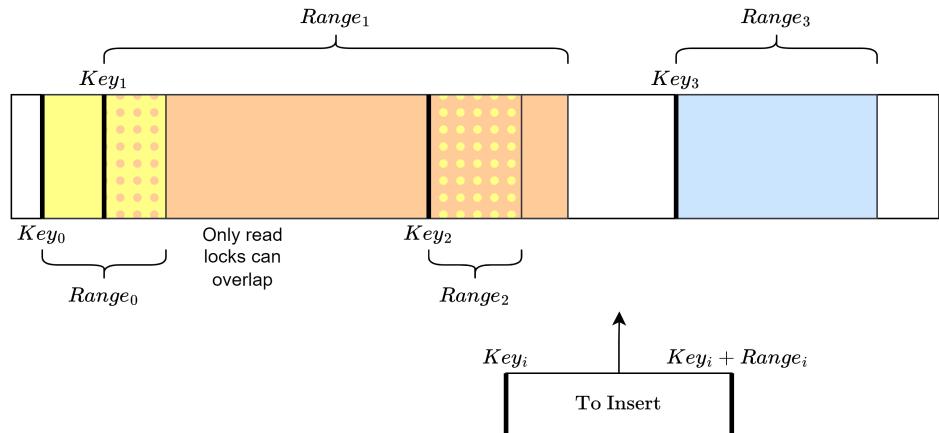
### Deadlocks

Deadlocks must be prevented, this can be achieved in several ways:

- Acquire locks in a global order → we cannot know which locks a query may need ahead of time.
- Complete a *dry-run* to determine required locks, before then accessing in global order → transactions may occur between *dry-run* and run, and hence change the set of locks required.
- Timeouts: Locks safe for a predefined time, after this if another transaction acquires the lock, it aborts the holding transaction.
- Cycle Detection: At regular intervals, inspect locks, waiters and holders to compute a graph, and abort transactions (usually youngest) to remove cycles.

### Lock Manager

$$locks[table] = \{Key_0 : (Range_0, Read), Key_1 : (Range_1, Read), Key_2 : (Range_2, Read), Key_3 : (Range_3, Write)\}$$



Manages the locking of ranges within tables as either read or write.

- Checks for conflicts in overlapping ranges
- Ensures locks are released properly

**UNFINISHED!!!**

**Serializable**  $2PL$  ensures realizability (and hence no anomalies).

**Deadlock Detection** Can be expensive to avoid & complex to implement.

**Mutual Exclusion** Ranges are locked, so cannot read & write, or write & write in parallel.

### 8.5.3 Timestamp Ordering

Each tuple is timestamped for *last read* and *last write*, and every transaction is timestamped at the start of execution.

**Read** Check tuple read timestamp (if newer than transaction timestamp abort), set read timestamp to transaction timestamp.

**Write** Check tuple write timestamp (if newer than transaction timestamp abort), set write timestamp to transaction timestamp.

### 8.5.4 Optimistic Concurrency Control (OCC)

Run transactions without locks, buffering reads, inserts and updates until commit. At commit check if the database is unmodified (if not then abort) and apply with locks.

- The simplest multiversion concurrency scheme.
- Can use a timestamp to determine when rows have been changed.

**Few Conflicts** Performant when the number of conflicts is low (e.g analytics database with few updates).

### 8.5.5 Multi-Version Concurrency Control (MVCC)

Store different versions of the tuple at different timestamps to allow a transaction to use old committed data, as new committed data is written to the *same* rows.

- Transactions use tuples that have the latest timestamp less than the transaction's start.
- Timestamps can be stored with tuples, or separately. Table structure needs to allow for multiple versions of the same *row* (e.g append new rows into one large table, or table entries are lists of rows etc.)

**Many Conflicts** Performs better than other concurrency control schemes when conflicts are frequent (conflicts do not force transactions to wait).

**Time travel** Multiple versions of tuples allow for quick rollbacks, to inspect recent past values for rows.

# Chapter 9

## Streams

### 9.1 Motivation

There are many data processing applications that deal with streams of relevance-priority data (e.g Sports data, weather data, telemetry (spacecraft, service usage)).

- Recent events are valuable, old events are not (and can be discarded or sent to data warehouse after some time)
- Users run a static query on an unbounded stream of data
- The state of the system must be bounded (limited memory)
- Timestamps for events are important (tradeoffs between performance, and accuracy), the order at which events are received is important.
- Results can be approximate

### 9.2 Push Operators

Rather than operators pulling in tuples (as in *volcano processing*), operators push tuples to the next stage.

```
// templated by the Event data type (for easy testing), would be some vector<variant<int, float, string,
template<typename Event>
class PushOperator {
public:
    virtual void process(Event data) = 0;
};
```

- As with volcano and bulk processing we can also send references to data (e.g indexes into a larger backing table) to avoid copies.
- Virtual method used to allow operators to be combined into queries at runtime.
- Can use `std::move` to reduce deep copying for large `Event` types (e.g vectors of variants).

#### 9.2.1 Naive Implementation

##### Output to Console

Some form of output operator is required to send data to the user (e.g player positions sent over the network to a live sports match website).

Here a basic Output operator pushes to a stream (e.g a file with `std::ofstream`, or to the console with `std::cout`).

```
template <typename Event>
class Output : public PushOperator<Event> {
    std::ostream &output_;
public:
    Output(std::ostream &output) : output_(output) {}

    void process(Event data) override { output_ << "→" << data << std::endl; }
};
```

## Selection

```
template <typename Event>
class Select : public PushOperator<Event> {
    PushOperator<Event> *plan_;
    std::function<bool(Event &) > predicate_;

public:
    Select(PushOperator<Event> *plan, std::function<bool(Event &) > predicate)
        : plan_(plan), predicate_(predicate) {}

    void process(Event data) override {
        if (predicate_(data)) plan_->process(std::move(data));
    }
};
```

## Project

Generalised here to just map a function over the stream.

```
// by default maps to same data type
template <typename InputEvent, typename OutputEvent = InputEvent>
class Project : public PushOperator<InputEvent> {
    PushOperator<OutputEvent> *plan_;
    std::function<OutputEvent(InputEvent)> function_;

public:
    Project(PushOperator<OutputEvent> *plan, std::function<OutputEvent(InputEvent)> function)
        : plan_(plan), function_(function) {}

    void process(InputEvent data) override {
        plan_->process(function_(std::move(data)));
    }
};
```

## Data Source

We also need to be able to pipe data directly into a chain of operators.

- Can implement a class to directly call `PushOperator::process`.
- Here a convenient interface is used to demonstrate terminal input.

```
template <typename Event>
class Source {
public:
    virtual void run() = 0;
};

template <typename Event>
class UserInput : public Source<Event> {
    PushOperator<Event> *plan_;
    std::istream &src_;

public:
    UserInput(PushOperator<Event> *plan, std::istream &src) : plan_(plan), src_{src} {}

    void run() override {
        for (Row r;; src_ >> r) plan_->process(std::move(r));
    }
};
```

## Combining Operators

We can then combine operators to form queries.

```
// Configure output
Output<int> console(std::cout);

// Build query
Project<int, int> mult(&console, [](auto i){ return i * 3; });
Select<int> even(&mult, [](auto &i){ return i % 2 == 0; });
UserInput<int> user(&even, std::cin);

// Get input stream
user.run();

1
2
->6
3
4
->12
```

### 9.2.2 PushBack

Resource usage of operators is important.

- Some operators may buffer rows (in order to resolve order, retain aggregates about current window (e.g min/max))
- Operators could be extracted to different threads, in which case some operators may run slowly compared with other operators.

One way to inform upstream operators about *backpressure* from pressured operators downstream is by returning some measure of pressure.

```
template<typename Event>
class PushOperator {
public:
    // return pressure on operator
    virtual float process(Event data) = 0;
};
```

Operators can then use some heuristic of time taken, buffer sizes and the *backpressure* from operators it pushes to.

## 9.3 Time

Systems often implicitly provide timestamps for pushed data, for example when joining data based on timestamps.

- Needs to be consistent (same stream results in the same output data).
- Needs to be performant/low overhead (reduce backpressure).

### Processing-Time

### Definition 9.3.1

Each operator timestamps data when it is pushed to the operator.

```
class SomeOperator : public Operator {
    // ... internal state
public:
    void process(InputEvent data) override {
        auto data_timestamp = std::chrono::system_clock::now();
        // ... use data & data_timestamp
    }
};
```

inconsistent | unpredictable | low-overhead

## Ingestion-Time

## Definition 9.3.2

Timestamp when received by the system (i.e the source object that pushes to the first operator).

```
class NetworkSource : public Source {  
    // ... internal state  
public:  
    void run() override {  
        for (;;) if (!network.buffer_empty()) {  
            auto data_timestamp = std::chrono::system_clock::now();  
            auto data = network.pop_next();  
            // ... use data & data_timestamp  
        }  
    }  
};
```

consistent | unpredictable | medium-overhead

## Event-Time

## Definition 9.3.3

Timestamps externally provided by the source supplying events to the data processing system as part of data input.

- System needs to ensure timestamps are ordered (external provider may not be correct).

```
class NetworkSource : public Source {  
    // ... internal state  
public:  
    void run() override {  
        for (;;) if (!network.buffer_empty()) {  
            auto data = network.pop_next();  
            // can just treat timestamp as normal data, or extract specially  
            auto data_timestamp = data.timestamp;  
            // ... use data & data_timestamp  
        }  
    }  
};
```

consistent | predicable | high-overhead

### 9.3.1 In-Order Processing

#### In-Order Processing

#### Definition 9.3.4

Events are assumed to be entered in timestamp order (or by some other monotonically progressing attribute - e.g counter).

- Greatly simplifies stream system implementation, a powerful guarantee.
- Difficult to ensure order guarantee holds (on a distributed, asynchronous system there is not global clock)

While it is usually prohibitively difficult to implement In-Order processing, we still need to have some guarantees on ordering for queries that rely on in-order data.

- If a single server is used to apply timestamps it can become a bottleneck.
- We can make some assumptions on bounds of how out-of-order messages can be received.
- We can reduce the *In-Order* to a *Sort-Order*.

#### Transactions

The stream of events is treated as a sequence of transactions.

- All events are inserted into persistent database

```
-- Store all inputs to persistent backing table
ON IncomingEvent newEvent INSERT INTO event_backing_table VALUES (newEvent)

-- Stream out data (e.g by selecting based on a predicate)
SELECT * FROM event_backing_table WHERE some_predicate(x, y, newEvent);
```

**Finite Memory** Streams are infinite, persistent database must have older entries cleared/garbage collected.

### Lateness Bounds

A lateness bound is assumed for any event, events outside this bound are dropped.

```
ON IncomingEvent newEvent INSERT INTO event_backing_table VALUES (newEvent)
SELECT * FROM event_backing_table WHERE some_predicate(x, y, newEvent)

-- Delete old data from the table using the new event's timestamp
DELETE FROM event_a_backing_table WHERE timestamp < (newEvent.timestamp - LATENESS_BOUND);
```

**Tune Lateness Bound** If the bound is too small (many tuples dropped), too large and memory resource becomes pressured by large backing table

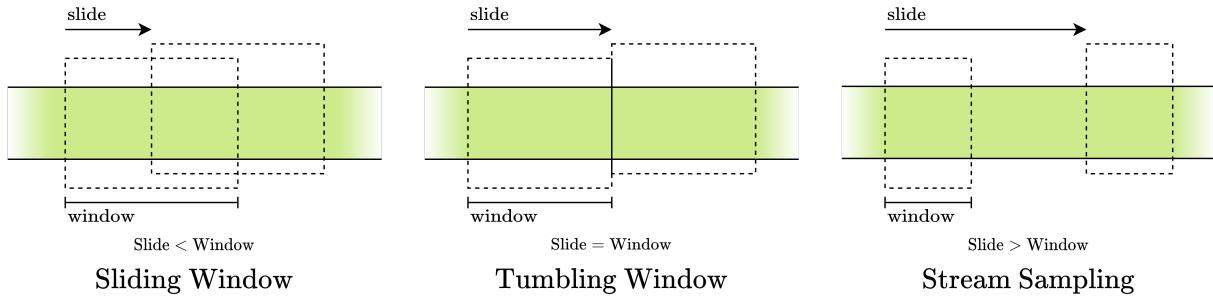
### Watermarks/Punctuation

The user sends a specific *punctuation* event to inform the system that all older events than a specific timestamp can be dropped.

```
ON IncomingEventWatermark e DELETE FROM event_backing_table WHERE timestamp < e.up_to_time;
```

**User Configurable** The user can specify when events should be dropped.

### 9.3.2 Windows



There are also *Session Windows* open and closed by an event (e.g user loggin in & out).

*Lateness bounds* are an implementation detail for ordering streams

*Windows* are SQL supported abstractions for viewing a slice of a stream, and are part of the language semantics.

#### SQL Windows

#### Extra Fun! 9.3.1

Despite being originally designed only for persistent databases, SQL added window functions in SQL 2003 (see changelog).

```

SELECT avg(temp) OVER (
    ORDER BY timestamp
    ROWS BETWEEN 5 PRECEDING AND 5 FOLLOWING
) AS smoothed_temp
FROM SpaceStationTemp;

```

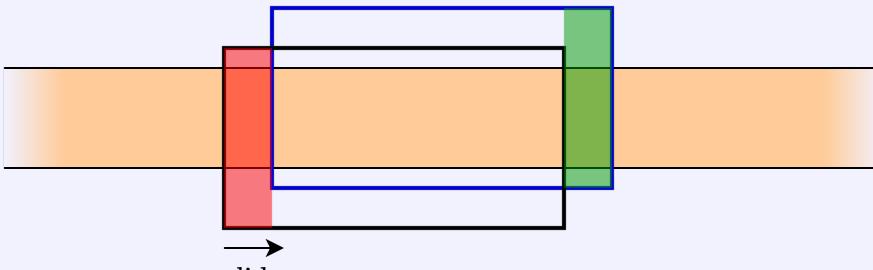
We can run aggregate functions on a window:

```

min, max, sum, count -- Distributive
avg                  -- Algebraic
percentile_cont      -- Holistic

```

Percentiles require the entire window to be read (cannot subdivide the window and combine as with `sum`, `count`).

Invertible Function	Definition 9.3.5
$\text{sum}(\text{new window}) = \text{sum}(\text{window}) - \text{old} + \text{new}$ 	

Functions with an inverse that can be used to remove rows sliding out of the window from the aggregation.

```

sum, count, avg      -- invertible
min, max, percentiles -- non-invertible

```

### 9.3.3 Aggregate Implementations

We can implement basic aggregate functions using the previous `PushOperator<Event>` abstraction.

#### Window Sum

```

class WindowSumAggregator : public PushOperator<float> {
    PushOperator<float> *plan_;
    std::vector<float> window_buffer_;
    size_t buffer_i_ = 0;
    float aggregate = 0;
    size_t count_ = 0;

    WindowSumAggregator(PushOperator<float> *plan, size_t windowsize)
        : plan_(plan), window_buffer_(windowsize) {}

    void process(float f) override {
        buffer_i_ = (buffer_i_ + 1) % window_buffer_.size();
        aggregate += f;
        count_++;
    }
}

```

```

        if (count_ > window_buffer_.size()) {
            aggregate -= window_buffer_[buffer_i_];
            window_buffer_[buffer_i_] = f;
            plan_->process(aggregate);
        } else {
            window_buffer_[buffer_i_] = f;
        }
    }
};


```

## Window Median

### Improve Me!

### Extra Fun! 9.3.2

The provided algorithm must copy the entire window for every `WindowMedianAggregator::process`. For large window sizes this is very slow, this can be made much more efficient!

```

class WindowMedianAggregator : public PushOperator<float> {
    PushOperator<float> *plan_;
    std::vector<float> window_buffer_;
    size_t buffer_i_ = 0;

    // for checking the window is filled
    size_t count_ = 0;

public:
    WindowMedianAggregator(PushOperator<float> *plan, size_t window_size)
        : plan_(plan), window_buffer_(window_size) {}

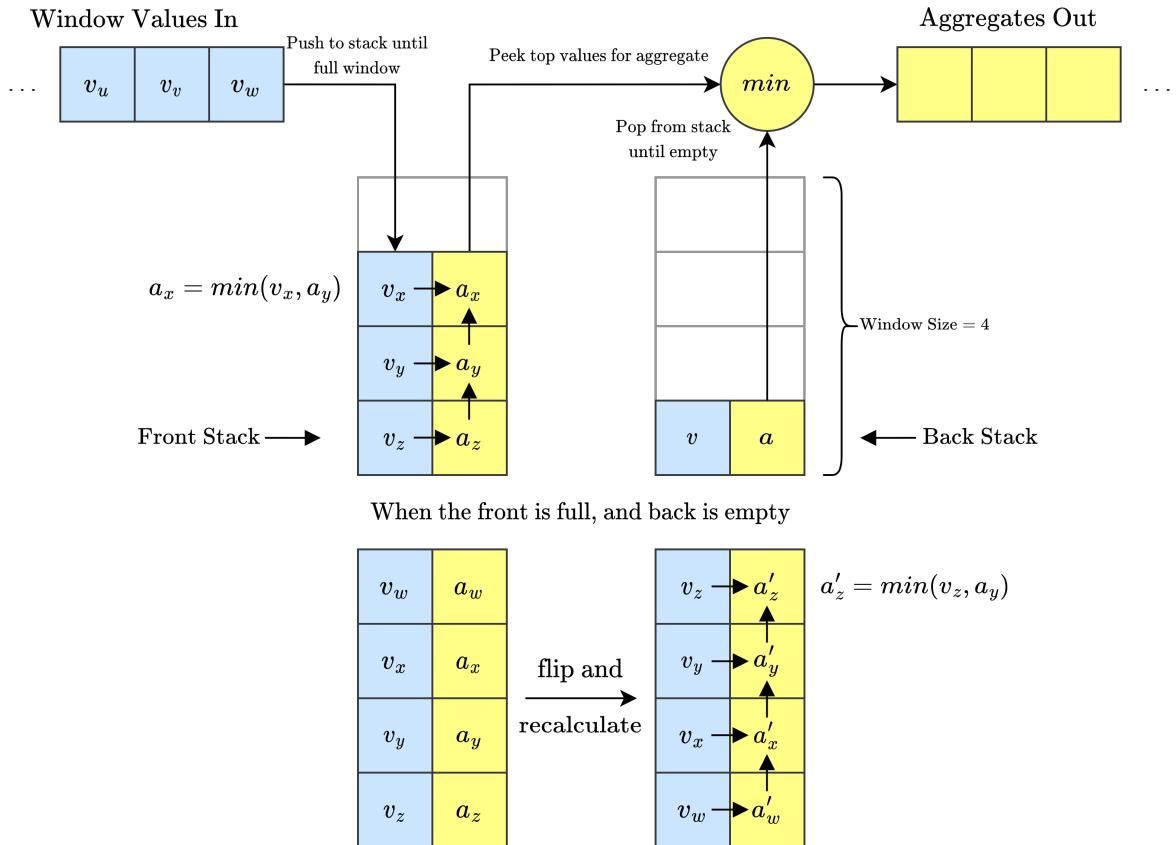
    void process(float f) override {
        const size_t size = window_buffer_.size();
        buffer_i_ = (buffer_i_ + 1) % size;
        window_buffer_[buffer_i_] = f;
        count_++;
        if (count_ > size) {

            // copy and sort, this can be made much more efficient using a multiset and vector
            // see multiset median trick: https://codeforces.com/blog/entry/68300
            std::vector<float> sorted = window_buffer_;
            std::sort(sorted.begin(), sorted.end());

            // if even size get average of two middle, else middle element
            if (size % 2 == 0) {
                plan_->process((sorted[size / 2] + sorted[(size / 2) - 1]) / 2);
            } else {
                plan_->process(sorted[size / 2]);
            }
        }
    }
};


```

### 9.3.4 Two Stacks Algorithm



Two stacks of max size *window size* are kept.

- Each contains aggregates calculated from below adjacent aggregates and current value.
- When the front stack is full, and back stack empty (occurs every  $\frac{1}{\text{window size}}$ ) flip the front stack, recalculate aggregates and set to back stack.

We can implement this using the previous `PushOperator<Event>` abstraction.

```
template <typename Event, Event agg(Event &, Event &)>
class WindowTwoStackAggregator : public PushOperator<Event> {
    PushOperator<Event> *plan_;

    // front stack
    std::vector<Event> front_values_;
    std::vector<Event> front_agg_;

    // back stack
    std::vector<Event> back_values_;
    std::vector<Event> back_agg_;

    // track the top of front and back stacks
    size_t window_pos = 0;

    // to determine when to start outputting aggregates
    size_t count_ = 0;

    // flip front stack to back stack, sets window_pos = 0
    // invariant: Must have window_size items present
    void flip() {
        size_t size = front_values_.size();
        assert(window_pos == size);
```

```

        for (size_t i = 0; i < size; i++) { back_values_[size - 1 - i] = front_values_[i]; }

        back_agg_[0] = back_values_[0];

        for (size_t i = 1; i < size; i++) { back_agg_[i] = agg(back_agg_[i - 1], back_values_[i]); }

        window_pos = 0;
    }

// Push an item to the front_stack, leaves the window_pos untouched
void push_front(Event r) {
    if (window_pos == 0) {
        front_values_[0] = r;
        front_agg_[0] = r;
    } else {
        front_values_[window_pos] = r;
        front_agg_[window_pos] = agg(r, front_agg_[window_pos - 1]);
    }
}

public:
    WindowTwoStackAggregator(PushOperator<Event> *plan, size_t window_size)
        : plan_(plan), front_values_(window_size), front_agg_(window_size),
          back_values_(window_size), back_agg_(window_size) {}

    void process(Event r) override {
        size_t max_size = front_values_.size();
        if (count_ < max_size) {
            push_front(r);
            window_pos++;
        } else {
            if (window_pos == max_size) { flip(); }

            push_front(r);
            plan_->process(agg(front_agg_[window_pos], back_agg_[max_size - 1 - window_pos]));
            window_pos++;
        }
        count_++;
    }
};

};


```

For example:

```

Output<int> console(std::cout);
WindowTwoStackAggregator<int, intmax> maxints(&console, 3);
UserInput<int> user(&maxints, std::cin);
user.run();

```

### Space Efficiency

### Extra Fun! 9.3.3

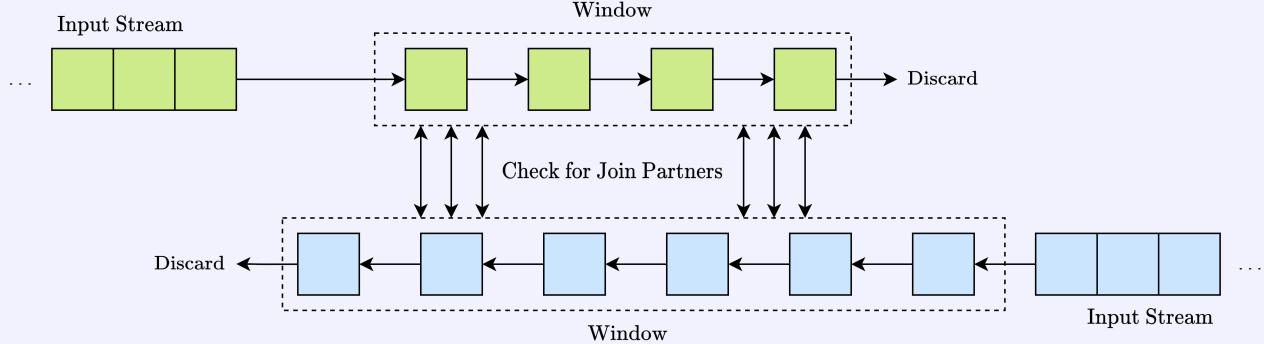
It is possible to implement the two-stacks algorithm more efficiently using a single vector (index from top down is back stack, bottom up is front stack).

## 9.4 Stream Joins

### 9.4.1 Handshake Join

#### Handshake Join

#### Definition 9.4.1

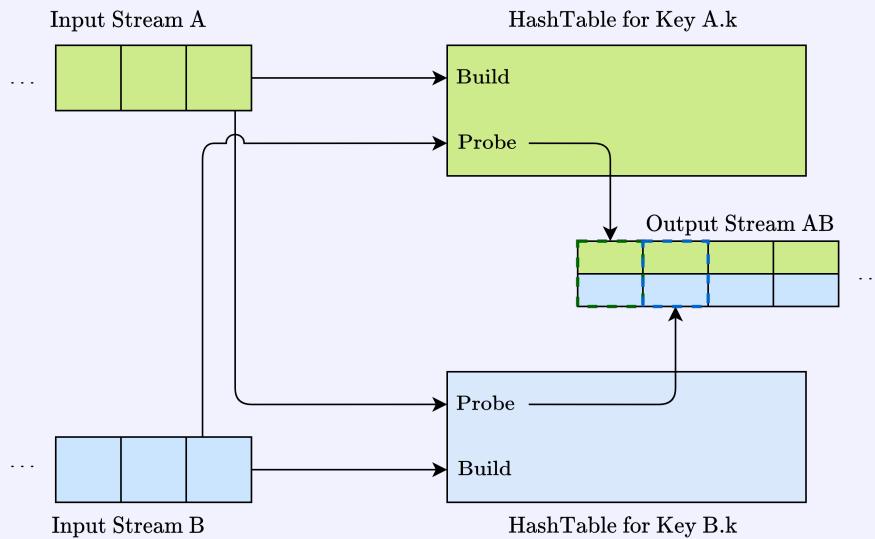


- A *nested loop join* joining over a window.
- Can be optimised for parallel window joins.
- Only works for window queries.

### 9.4.2 Symmetric Hash-Joins

#### Symmetric Hash-Joins

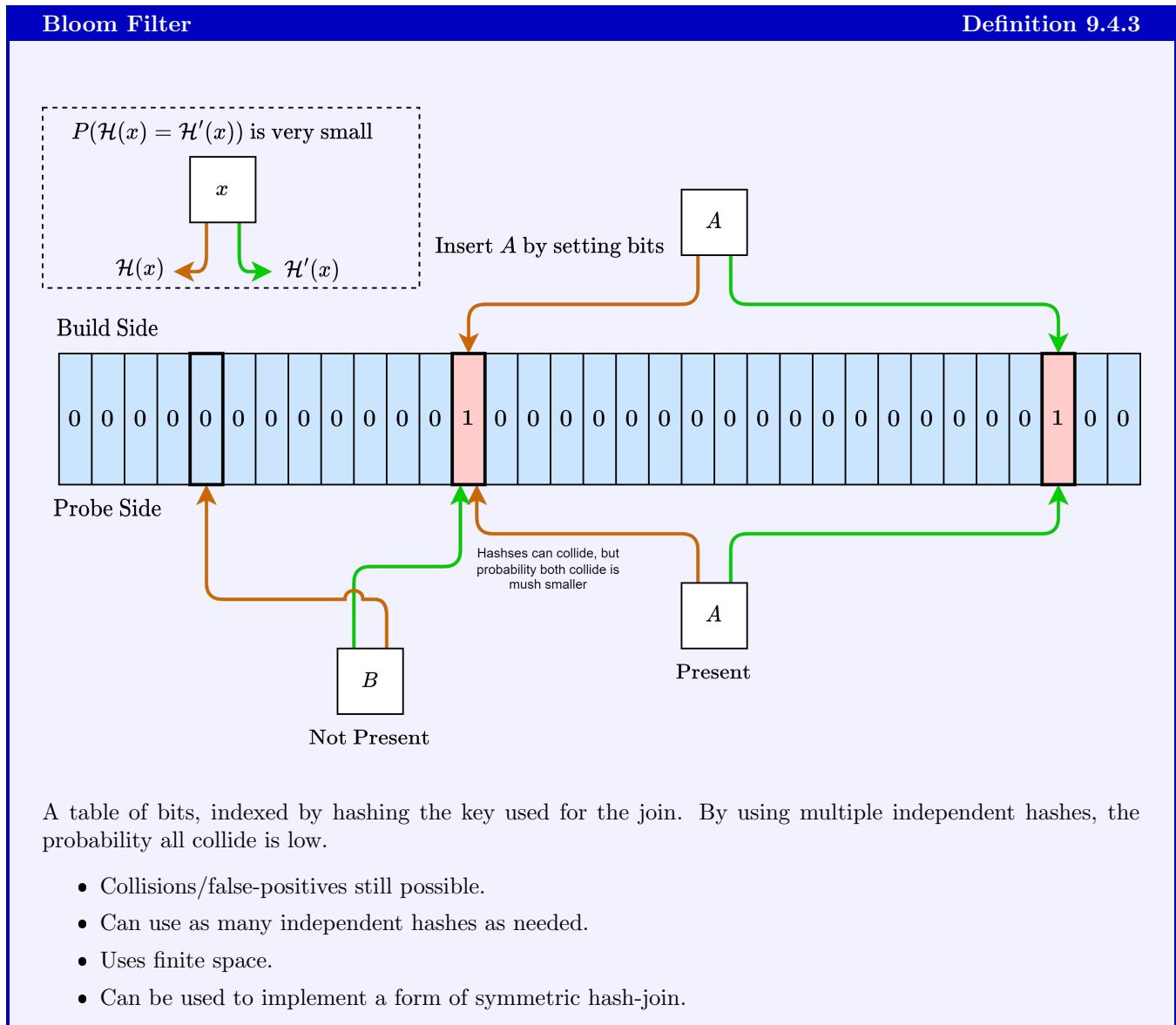
#### Definition 9.4.2



Both input streams build their own hashtable, while probing the other. Matches from probes are inserted into the joined output stream.

- A *pipelineable* hash join
- Does not have an equivalent window oriented version
- Hashtables grow with unbounded input streams, so needs some form of garbage collection of older / not joinable hashtable entries.

### 9.4.3 Bloom Filters



### Tuning Bloom Filters

Bloom filters have several parameters that can be tuned.

$m$  bits in filter

$n$  expected number of distinct elements

$k$  number of hash functions

$\epsilon$  False-positive rate

$$m \cong -1.44 \times n \times \log_2(\epsilon) \quad k \cong \frac{m}{n} \times \log_e(2) \quad \epsilon = \left(1 - e^{-\frac{k \times n}{m}}\right)^k$$

## Chapter 10

# Advanced Topics

**UNFINISHED!!!**

# Chapter 11

## Credit

### Image Credit

Front Cover OpenAI Dall-E.

### Content

Based on the excellent Data Processing Systems course taught by Dr Holger Pirk.

Includes content from the first year databases course 40007 by Dr Peter McBrien.

These notes were written by Oliver Killane.