60029

Data Processing Systems
Imperial College London

Contents

Chapter 1

Introduction

1.1 Logistics

Dr Holger Pirk

The module is taught by Dr Holger Pirk.

- Taught as prerecorded lectures with in-person Q&As
- Weekly tutorial sheets covered during the Q&A sessions
- One coursework in teams of 3 (with a competition!)

1.1.1 SQL

SQL is not prerequisite for this course!

The **40007 - Introduction to Databases** module covers all that is required. This module is about the implementation of data processing systems, not using databases & SQL.

1.1.2 C++

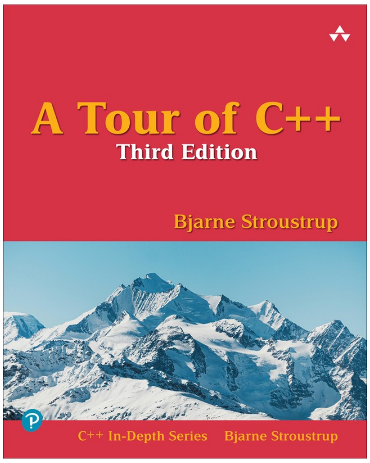
C++ is not prerequisite for this course!

This course contains many code examples in C++. This course represents a great opportunity to learn at least a small part of its enormity, and to apply some of this in the coursework.

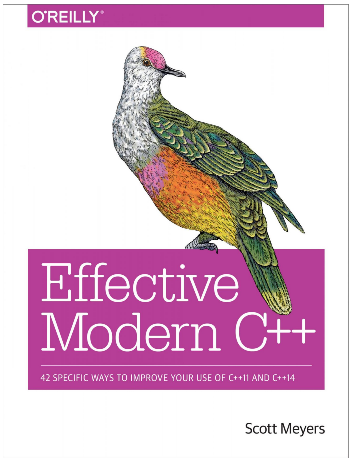
> **C++ > C with classes** — ***Extra Fun!* 1.1.1**
>
> C++ was originally developed around 1983 by Bjarne Stroustrup as *'C with classes'* and implemented as a transpiler to C called cfront.

CppReference
Comprehensive C++ and standard template library documentation.

Mastering CMake
CMake Documentation



A tour of C++
An overview of C++, only need C knowledge carried from pintos



Effective Modern C++
modern C++ exploration & best practices

The above books are available online from O'Reilly through Imperial (use institution login) for free & you may find useful for learning more C++ (though beyond the understanding required for this module).

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)

Storage	Needs to be persistent (but also needs to be fast)
Data Ingestions	Needs to allow for easy import of data (e.g by providing a csv, another database's url)
Concurrency	To exploit parallelism in hardware (e.g multithreaded, distributed over several machines)
Data Analysis	For inspection (typically the reason to hold data in for first place)
User Defined Functions	Needs to allow users to write their own queries on data.
Standardized Programming Model	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.
Access Control	Not all data is shared between all users.
Self-Optimization	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 the acquires, stores and processes a significant amount of information. Core functionality of the application is based on data.

Online Transaction Processing (OTP) — **Definition 1.3.2**

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

Goal: **Throughput**

Online Analytical Processing (OLAP) — **Definition 1.3.3**

Running a single data analysis task. Mixture of query types. Queries are ad-hoc.

Goal: **Latency**

Reporting — **Definition 1.3.4**

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 *Reporting* pattern being used in industry.

- A supermarket getting the day's sales, and stock-take.
- A trading firm computing their position and logging the days trades at market-close and informing regulators, clearing, risk department.
- A company's payroll systems running weekly using week long timesheets.

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.

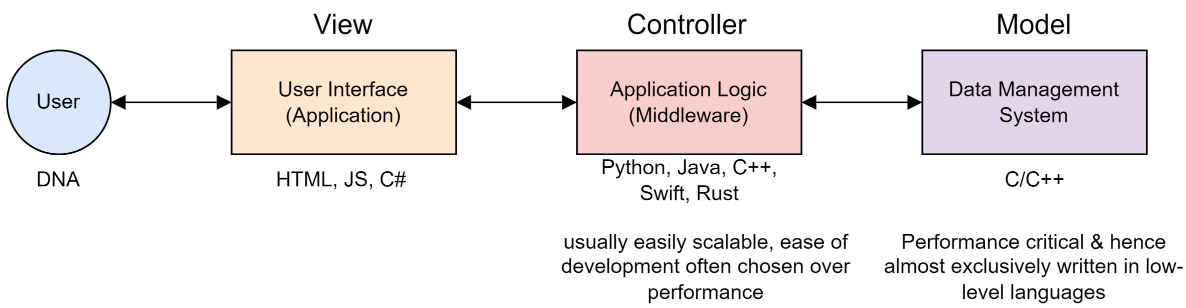
Data-Intensive Applications can be differentiated from *Data Management Systems* (though there is ample ambiguity):

- 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 1.3.5**

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

Efficiency Ideally should be as fast as a bespoke, hand-written solution.

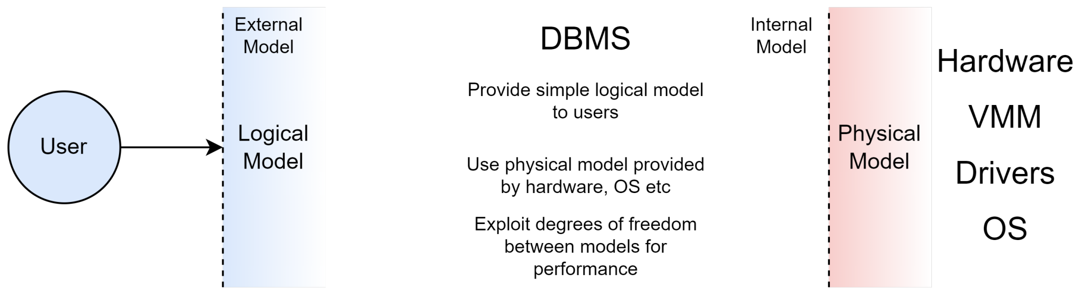
Resilience Must be able to recover from failures (software crashes, power failure, hardware failure)

Robustness Predictable performance (semantically small change in query $\Rightarrow$ similarly small change in performance)

Scalability Can scale performance with available resources.

Concurrency Can serve multiple clients concurrently with a clear model for how concurrency will affect results.

1.4.2 Logical/Physical Data Model Separation



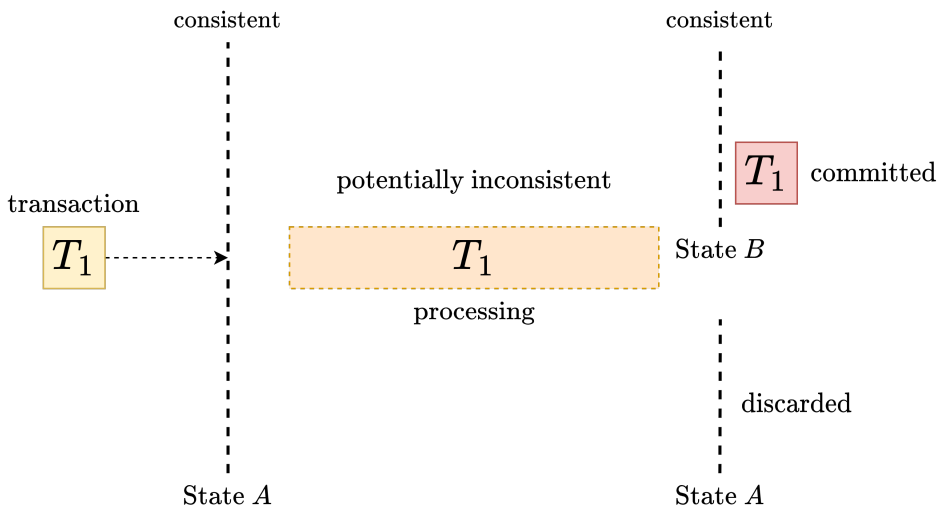
1.4.3 Transactional Concurrency

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.

Atomic	A transaction either runs entirely (and is committed) or has no effect. (All or nothing)
Consistent	A transaction can only bring the database from one valid (for some invariants) state to another. Note that there may be inconsistency between.
Isolated	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).
Durable	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 *two-phase locking*.
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.

Concurrency control is discussed in detail in chapter 7.

1.4.4 Read Phenomena

You should already be familiar with some basic anomalies/phenomena, these include:

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

Discussed in detail in chapter 7.

Serialisable — **Definition 1.4.5**

Dirty Read	*Non-repeatable Read*	*Phantom Read*
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 `SELECT ... FROM ... WHERE ... ;` to avoid *phantom reads*.
- Prevents all 3 read phenomena and is the strongest isolation level.

Repeatable Reads — **Definition 1.4.6**

Dirty Read	*Non-repeatable Read*	*Phantom Read*
Prevented	Prevented	Allowed

- Unlike *serialisable* 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**

Dirty Read	*Non-repeatable Read*	*Phantom Read*
Prevented	Allowed	Allowed

Mutual exclusion is held for writes, but reads are only exclusive until the end of a `SELECT ... ;` 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**

Dirty Read	*Non-repeatable Read*	*Phantom Read*
Allowed	Allowed	Allowed

The weakest isolation level and allows for all *read phenomena*.

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 & SQL derived languages).

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.

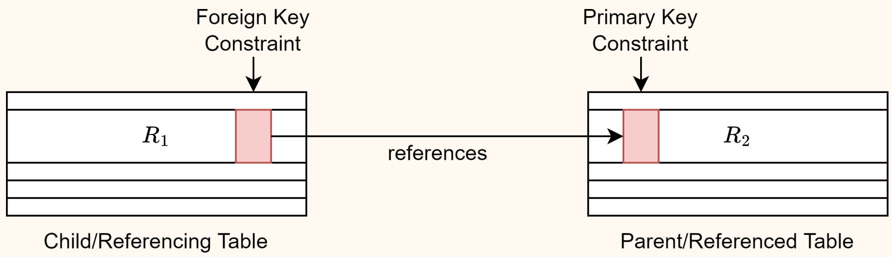
```
CREATE TABLE foo (bing INTEGER, zog TEXT, bar INTEGER);
```

- Integrity constraints (foreign keys, nullability, uniqueness etc)

```
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:

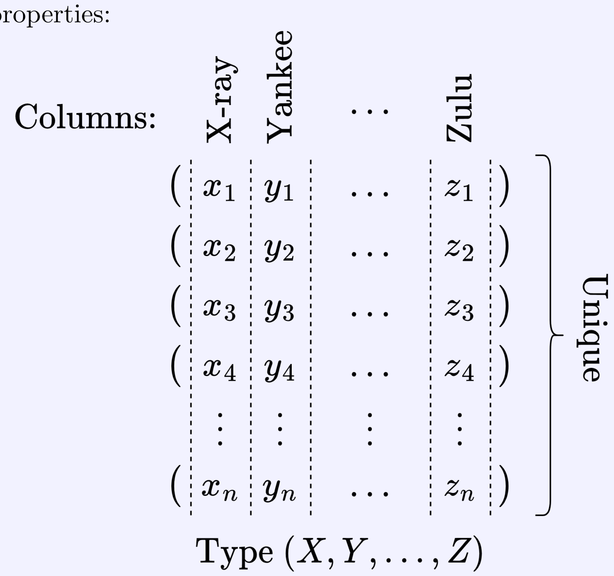
Vector	Ordered collection of objects (same type)
Tuple	Ordered collection of objects (can be different types)
Bag	Unordered collection of objects (same type)
Set	Unordered collection of unique objects (same type)

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).



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

Project	**Select**	**Cross/Cartesian product**	**Union**	**Difference**
π	σ	$\times$	$\cup$	$-$ or $\setminus$

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 to get the names of every book ordered by a current Customer in relational algebra and SQL (you may ignore differences due to bag vs set semantics).

```
CREATE TABLE Book (
  BookID INTEGER NOT NULL,
  Title  VARCHAR(20),
  Author VARCHAR(20),
  ISBN   VARCHAR(13)
);

CREATE TABLE OrderedItem (
  OrderID INTEGER NOT NULL,
  BookID  INTEGER NOT NULL
);
```

```
CREATE TABLE Order (
  OrderID    INTEGER NOT NULL,
  CustomerID INTEGER NOT NULL,
  Price      DECIMAL(18,2)
);

-- Stores current customers
CREATE TABLE Customer (
  CustomerID      INTEGER NOT NULL,
  ShippingAddress VARCHAR(50),
  Name            VARCHAR(20)
);
```

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

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

Note that this will produce duplicates (bag semantics), we can remove these using a `SELECT DISTINCT`.

Unique Addresses **Example Question 2.1.3**

Using the previous schema create a query to get each author that has only sold to one address (can potentially ship to multiple customers at the same address)

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

Here a natural join is:

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

```
SELECT Book.Author
FROM (
  SELECT Book.Author, Customer.ShippingAddress
  FROM ((Customer NATURAL JOIN Order) NATURAL JOIN OrderedItem) NATURAL JOIN Book
)
GROUP BY Book.Author
HAVING COUNT(*) = 1;
```

2.1.2 Nomenclatures

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

2.1.3 Schemas

Database Schema **Definition 2.1.3**

The logical structure of the database that is exposed to users (e.g through SQL).

- defines the tables, their columns, relations, indexes and constraints in a database.
- In some systems (e.g postgres), can include permissions/access control, functions, views and more.
- Analogous to the *type* of the database.
- Does not describe the physical layout,

```
-- one postgres database can have many different schemas
CREATE SCHEMA my_schema;

CREATE TABLE foo(
  id SERIAL PRIMARY KEY,
  name VARCHAR(20) NOT NULL,
  bestie SERIAL REFERENCES foo(id),
  added_date DATE NOT NULL CHECK (added_date > '2000-01-01')
);

CREATE INDEX idx_added_date_desc ON people (added_date DESC);
```

2.2 Implementing Relational Algebra in C++

A note on types... — ***Extra Fun!* 2.2.1**

Here we will express operators & relations in the C++ type system.

In real databases the schema & types are not know when the database itself is compiled, but rather later at runtime (i.e do not know the types of columns, tables until they are created, amended, and operated on at runtime).

In order to implement a model of relational algebra 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 *variadict 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...> {};
```

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

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

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

We can first create a projection.

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

```cpp
  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...>(typename InputOperator::OutputType)>,
          set<pair<string, string>>>
      projections;

  // Constructor for function application
  Project(InputOperator input,
          function<tuple<outputTypes...>(typename 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!* 2.2.2**

> 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.

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

struct Column {
  string name;

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

// 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) {}
};

template <typename InputOperator>
struct Select : public Operator<typename InputOperator::OutputType> {
  InputOperator input;
  Condition condition;
```

```
  Select(InputOperator input, Condition condition) : input(input), condition(condition) {};
};
```

Enums vs Enum classes — *Extra Fun!* 2.2.3

`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.

In order to concatenate the types of the product relations we create `Concat<left types..., right types...>`.

```
// 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;
```

Template Magic — *Extra Fun!* 2.2.4

If you are interested in how this works, see cppreference - template specialisation.

```
// 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

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

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 (know 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){};
};
```

Have a go! ***Extra Fun!* 2.2.5**

The provided examples are only one way to mock up / sketch relation algebra with C++. Consider:

- Using runtime polymorphism `virtual` methods in derived operator classes and `std::variant` data types, rather than using templates.
- Going all-in on compile time programming with `constexpr`.
- Using `concepts` / `requires` to constrain the operator types.

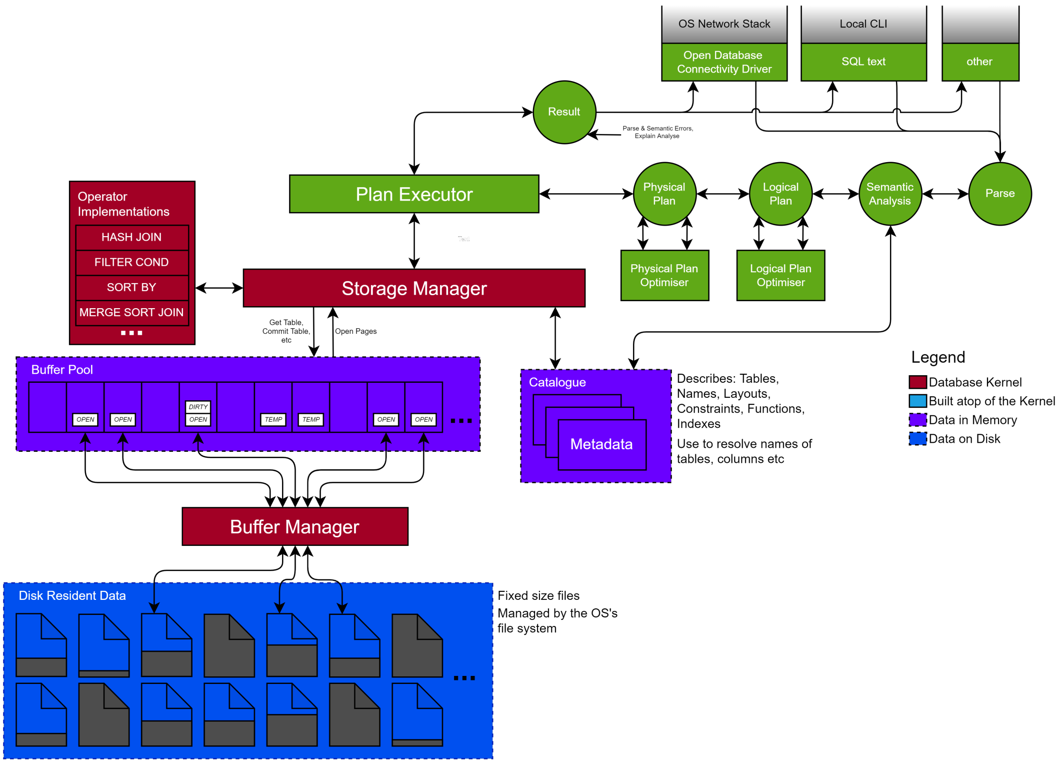
Chapter 3

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 Buffer Manager

Buffer Manager — **Definition 3.2.1**

Part of the database kernel that manages disk-resident data, and moves disk resident data required by the storage manager into pages in memory (the buffer pool).

3.2.2 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.

Optimising for Cache — ***Extra Fun!* 3.2.1**

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

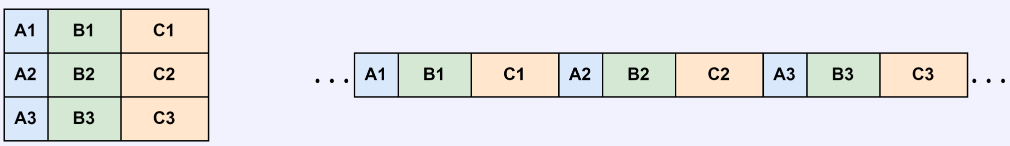
Locality — **Definition 3.2.2**

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.

Spatial	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.
Temporal	Accessing the same location.	Lines stay until evicted due to capacity or flush, load-store queues effectively cache resent accesses.

N-ary Storage — **Definition 3.2.3**

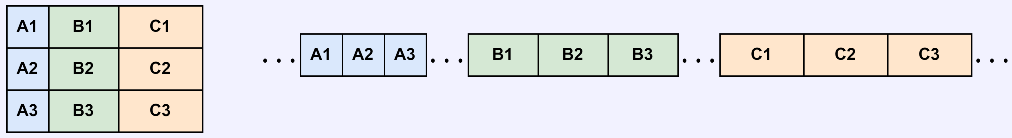
Tuples are stored adjacently.



- 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 3.2.4**

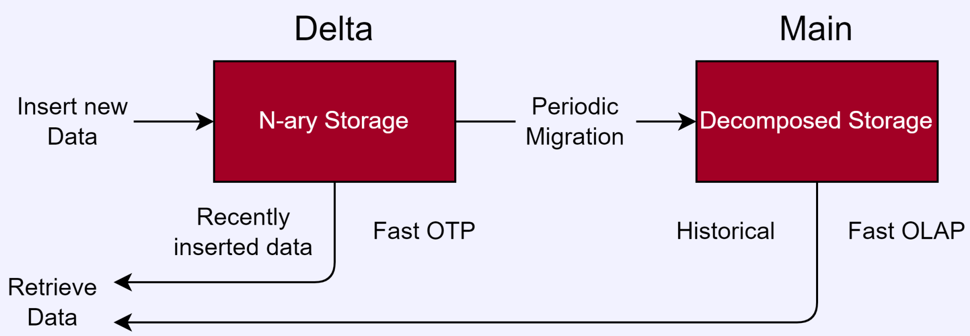
Each field of the tuple is stored in a separate array.



- 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 3.2.5**

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).

3.2.3 Catalog

Catalog **Definition 3.2.6**

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

Dense **Definition 3.2.7**

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.

3.2.4 Disk Storage

Disks differ from main memory:

Larger Pages Disks accessed in blocks. Main memory is to disk what cache is to main memory, but with lines (pages) on the oder of kilobytes. For *N-Ary* storage each page behaves like a mini-database.

Higher Latency Order of milliseconds rather than micro/nano seconds (main memory/cache).

Lower Throughput Megabytes per second versus gigabytes per second for main memory.

Accessed Through OS Programs must interact with OS for every read and write from disk through an OS provided file abstraction (this can be negated & there are exceptions). File size is limited, the DBMS must manage a pool of used files and determine offsets to access data contained.

As a result disk IO often dominates DBMS costs. Small reductions from complex IO management strategies are often more significant than any overhead they incur.

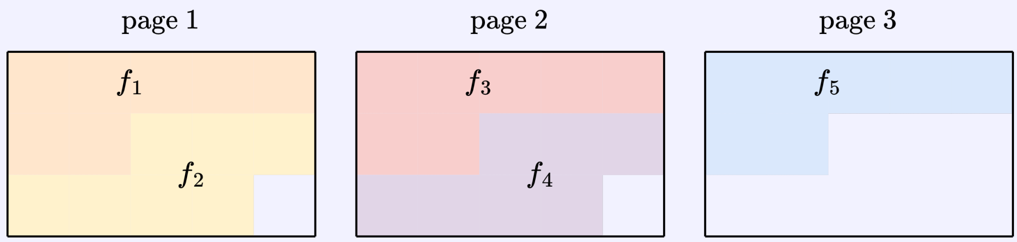
Buffer Manager **Definition 3.2.8**

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 3.2.9**

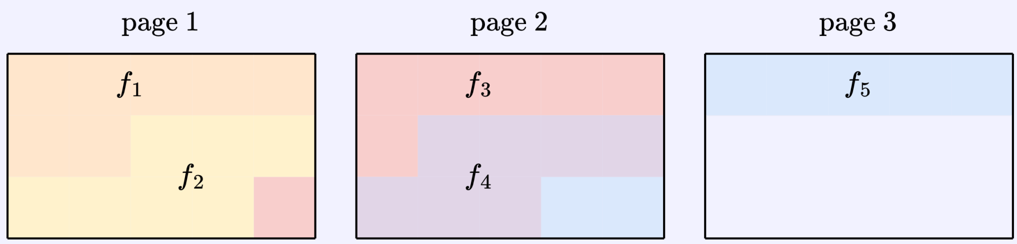
Records only allocated on the page is there is space.



- 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 know 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 3.2.10**

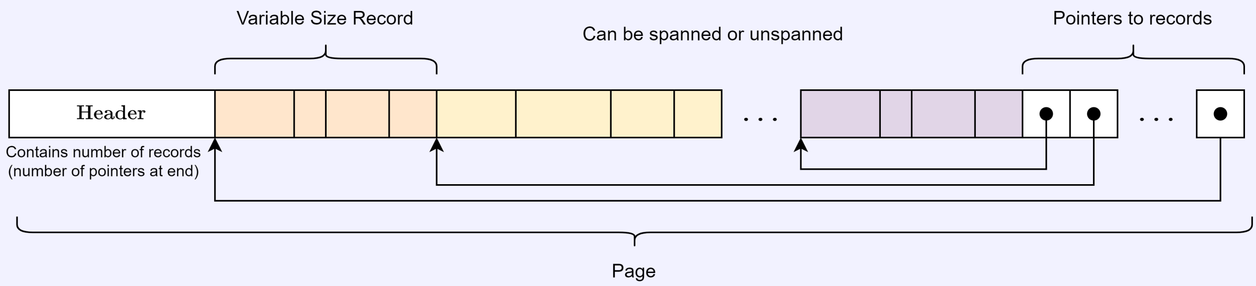
Records placed across page boundaries.



- 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 3.2.11

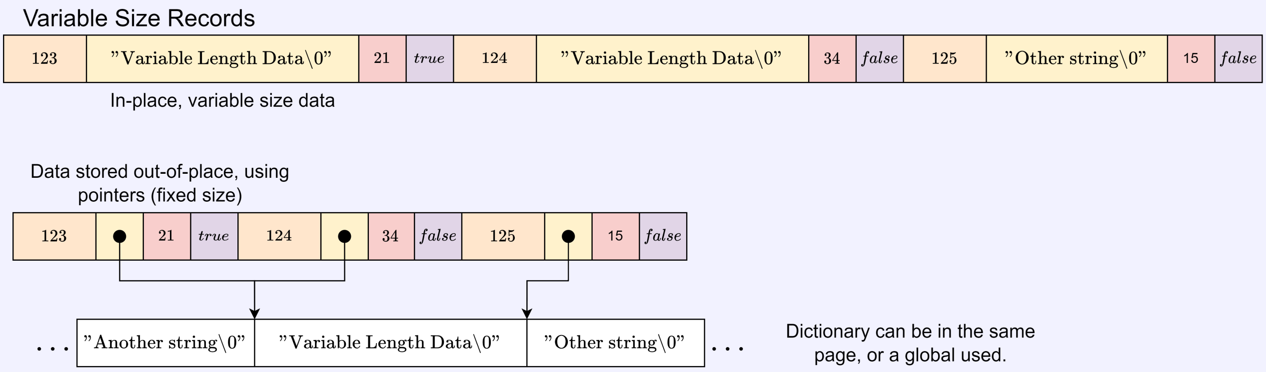
To allow faster/constant time lookup for variable size records.



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.

Dictionaries — Definition 3.2.12

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

Chapter 4

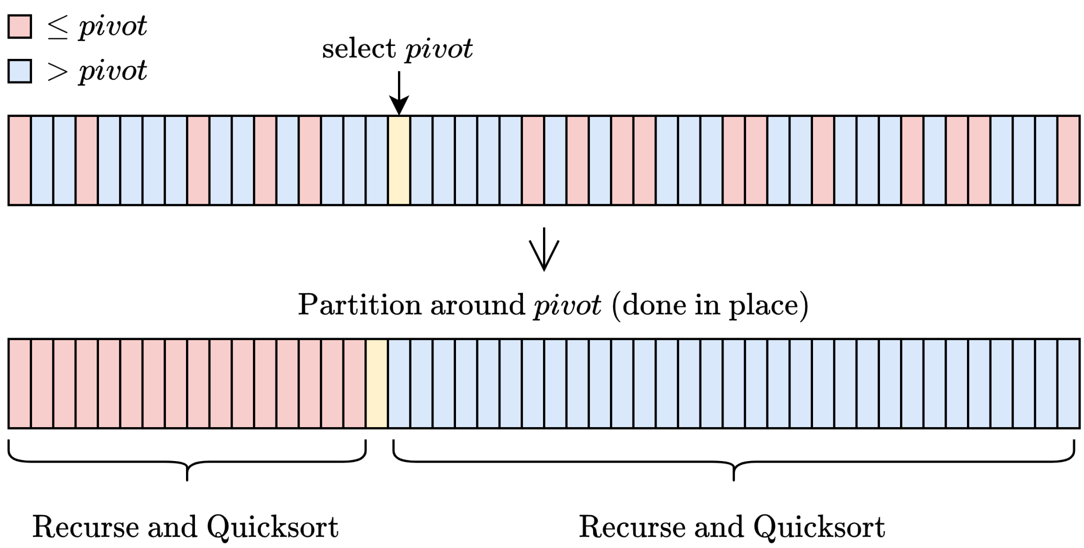
Algorithms and Indices

Play & Contribute! ***Extra Fun!* 4.0.1**

C++ implementations, tests and benchmarks for this section are included in the associated code directory for this chapter.

4.1 Sorting Algorithms (unassessed)

4.1.1 Quicksort




```
#pragma once
#include <vector>

using namespace std;

template <typename T, bool comp(const T&, const T&)>
size_t partition(vector<T>& sort_vec, size_t start, size_t end) {
    // select pivot
    T pivot = sort_vec[start];

    // find number of items before pivot ()
    size_t count = 0;
    for(size_t i = start + 1; i < end; i++) {
        if (comp(sort_vec[i], pivot)) count++;
    }
```

```
    // move pivot to its final position
    size_t pivotIndex = start + count;
    swap(sort_vec[pivotIndex], sort_vec[start]);
    size_t i = start, j = end - 1;

    // partition by finding pairs of elements that can be swapped around the pivot
    while(i < pivotIndex && j > pivotIndex) {
        while(comp(sort_vec[i], pivot)) i++;
        while(!comp(sort_vec[j], pivot)) j--;

        if(i < pivotIndex && j >= pivotIndex) {
            swap(sort_vec[i], sort_vec[j]);
            i++;
            j--;
        }
    }

    return pivotIndex;
}

template <typename T, bool comp(const T&,const T&)>
void quicksort_helper(vector<T>& sort_vec, size_t start, size_t end) {
    if(start + 1 >= end) return;

    size_t p = partition<T, comp>(sort_vec, start, end);
    quicksort_helper<T, comp>(sort_vec, start, p);
    quicksort_helper<T, comp>(sort_vec, p + 1, end);
}

template <typename T, bool comp(const T&,const T&)> void quicksort(vector<T>& sort_vec) {
    quicksort_helper<T, comp>(sort_vec, 0, sort_vec.size());
}
```

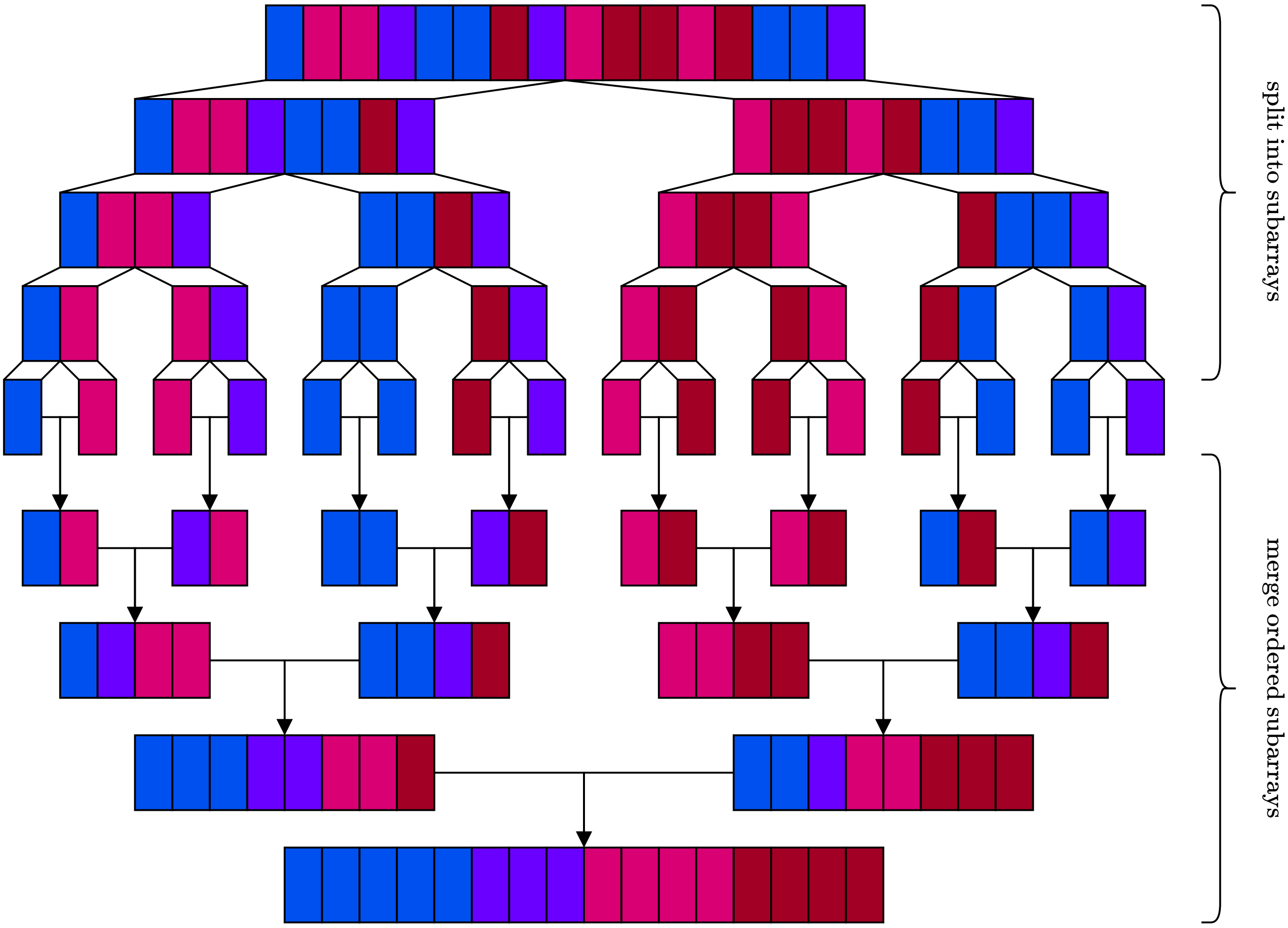
Average Complexity	Worst-Case Complexity
$O(n \log n)$	$O(n^2)$

Selecting a balanced pivot (ideally the median) is important to avoid worst-case complexity (where all others are larger or smaller than the pivot). Sampling multiple possible pivots negates this, as do other hybrid sorts.

Quick On average typically faster than merge or heapsort.
In-Place Sort can be performed entirely in place (no extra memory required, good temporal locality).
Parallel Is trivial to parallelise.

Worst-Case Avoiding $O(N^2)$
Blind Does not take advantage of partially sorted data (in fact this can lead to worst-case depending on pivot selection).

4.1.2 Merge Sort




```
#pragma once
#include <vector>
#include <iostream>

using namespace std;

template<typename T, bool comp(const T&,const T&)>
vector<T> mergesort_helper(const vector<T>& unsorted, size_t start, size_t end) {
    if (end - start == 1) {
        return {unsorted[start]};
    } else {
        // determine split
        const size_t mid = (start + end) / 2;

        // recursively mergesort
        const vector<T> left_split = mergesort_helper<T, comp>(unsorted, start, mid);
        const vector<T> right_split = mergesort_helper<T, comp>(unsorted, mid, end);

        // merge splits
        vector<T> merged;
        merged.reserve(end - start);

        // add largest to merged vector until one split is empty
        size_t left_ptr = 0;
        size_t right_ptr = 0;
        while (left_ptr < left_split.size() && right_ptr < right_split.size()) {
            const T& left = left_split[left_ptr];
            const T& right = right_split[right_ptr];
```

```
            if (comp(left, right)) {
                merged.push_back(left);
                left_ptr++;
            } else {
                merged.push_back(right);
                right_ptr++;
            }
        }

        // add empty split to vector, here only one loop iterates
        while (right_ptr < right_split.size()) {
                merged.push_back(right_split[right_ptr]);
                right_ptr++;
        }
        while (left_ptr < left_split.size()) {
                merged.push_back(left_split[left_ptr]);
                left_ptr++;
        }

        return merged;
    }
}

template<typename T, bool comp(const T&, const T&)> void mergesort(vector<T>& unsorted) {
    if (unsorted.size() > 0) {
        vector<T> sorted = mergesort_helper<T, comp>(unsorted, 0, unsorted.size());
        swap(sorted, unsorted);
    }
}
```

Average Complexity	**Worst-Case Complexity**
$O(n \log n)$	$O(n \log n)$

Worst-Case Same time complexity as best-case.
Parallel Is trivial to parallelise.

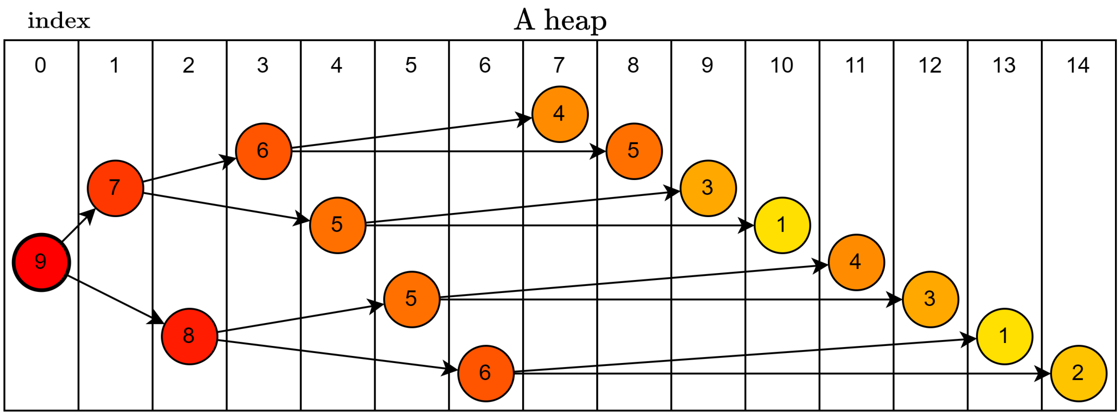
Out-of-Place Extra memory required for merges / cannot be done entirely in place & worse locality.
Blind Does not take advantage of partially sorted data.

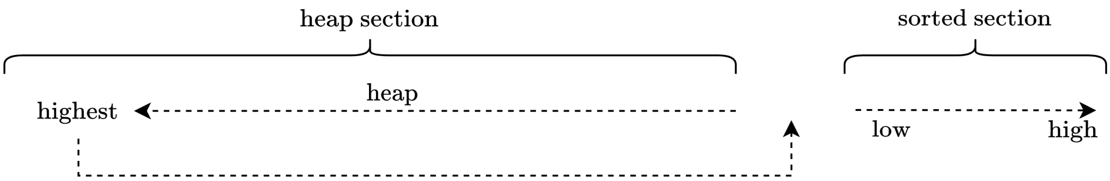
Bring order to the galaxy! ***Extra Fun!* 4.1.1**

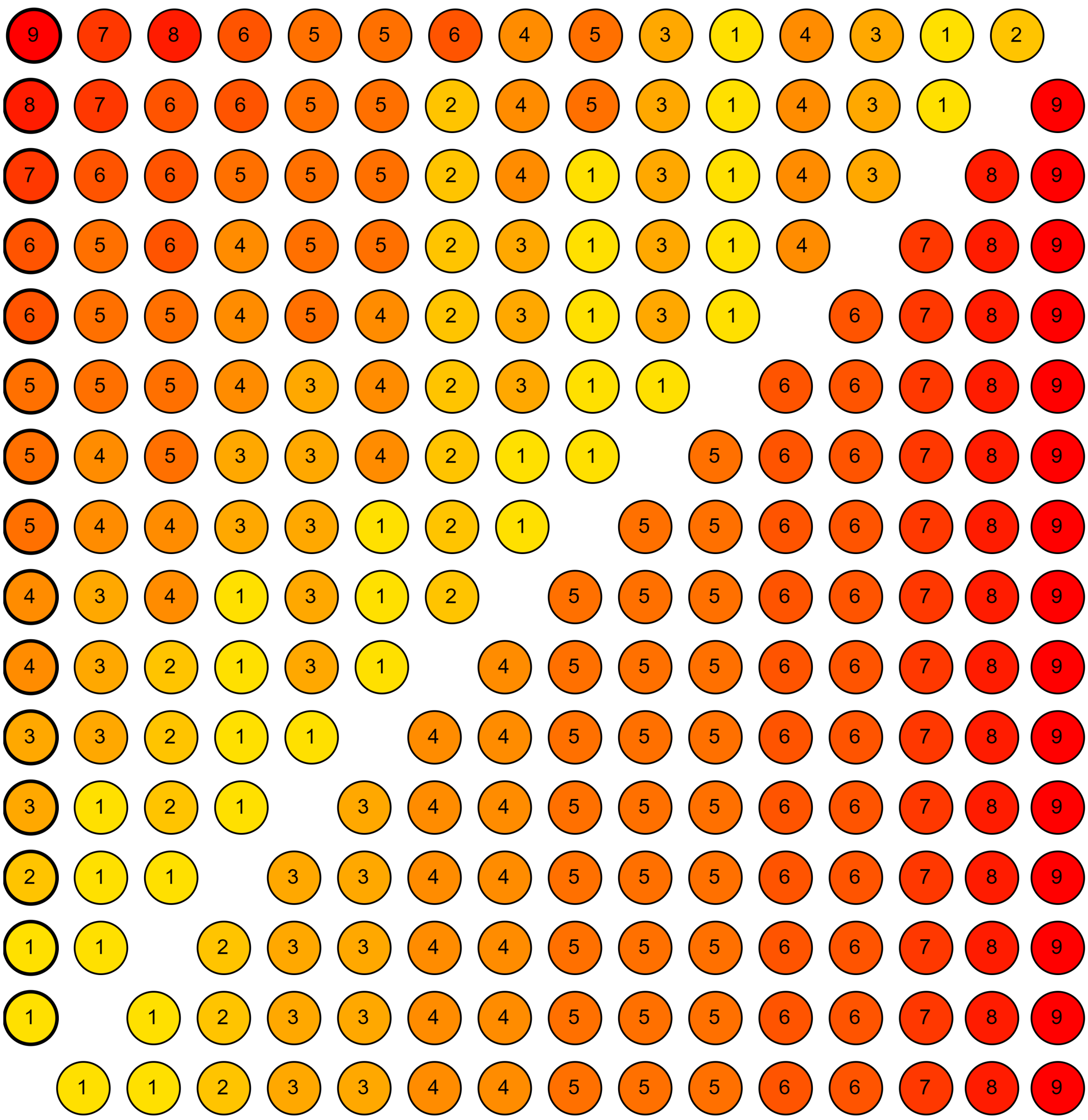
The provided simplistic mergesort allocates many vectors for out-of-place merging. Try and improve it!

- Reuse one scratch vector for out-of-place merging
- Don't copy the single element *splits* at the merge sort's base case.
- Predicate the merge process - see subsection 9.3.3.

4.1.3 Heapsort







```cpp
#pragma once
#include <vector>

using namespace std;

// INV: we have a heap from heap[root+1:]
template<typename T, bool comp(const T&,const T&)>
void siftDown(vector<T>& heap, size_t root, size_t end) {
    for (;;) {
        size_t largest = root;
        size_t left_root = 2 * root + 1;
        size_t right_root = 2 * root + 2;

        if (left_root < end && comp(heap[largest], heap[left_root]))
            largest = left_root;

        if (right_root < end && comp(heap[largest], heap[right_root]))
            largest = right_root;

        if (largest != root) {
            swap(heap[largest], heap[root]);
            root = largest;
        } else {
            break;
        }
    }
}


template<typename T, bool comp(const T&,const T&)>
void heapsort(vector<T>& unsorted) {
    // Create a heap structure (parent is larger than both children)
    for (size_t i = unsorted.size(); i > 0; i--) {
        siftDown<T, comp>(unsorted, i - 1, unsorted.size());
    }

    // Take each element, and re-siftDown the heap
    // - unsorted[0:i] is a heap
    // - unsorted[i:] is sorted
    for (size_t i = unsorted.size(); i > 1; i--) {
        swap(unsorted[0], unsorted[i - 1]);
        siftDown<T, comp>(unsorted, 0, i - 1);
    }
}
```

Average Complexity	**Worst-Case Complexity**
$O(n \log n)$	$O(n \log n)$

Worst-Case Same time complexity as best-case.

Top-N If we only want to n largest items, heapsort can extract this without sorting the entire array.

In-Place Sort can be performed entirely in place (no extra memory required, good temporal locality).

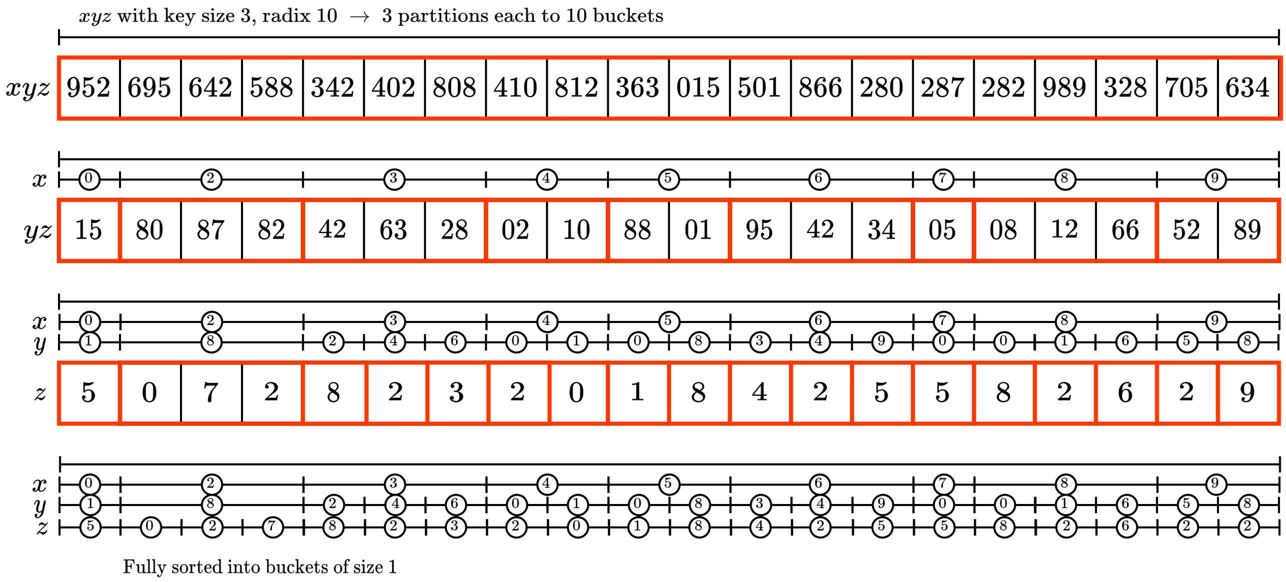
Parallelism Difficult to parallelize.

Blind Does not take advantage of partially sorted data.

Bring order to the galaxy! ***Extra Fun!* 4.1.2**

There are parallelised heapsort implementations, such as https://arxiv.org/ftp/arxiv/papers/0706/0706.2893.pdf.

4.1.4 Radix Sort



A non-comparative sorting algorithm. Rather than comparing elements to determine an order, partition into buckets based on some key (e.g digit).

Worst-Case Complexity
$O(\text{key width} \times \text{radix})$

Parallelism	Trivially parallel.
In-Place	In-place variants of the sort exist.

Inefficient on Small Data	Often paired with another sort (e.g insertion sort) used when buckets are small.
Lexicographical Only	Data must have a lexicographical order (e.g integers)

4.1.5 Hybrid Sorts

Sorts can be combined to avoid worst-case weaknesses. A typical pattern is to pair quicksort with another sort switch to to avoid quicksort's poor worst-case complexity.

Pattern Defeating Quicksort	A hybrid sort using quicksort's fast average complexity, and heapsort's good worst-case complexity.
Multi-Pivot Quicksort	For improved cache performance (a dual-pivot quicksort was used in Java 7).
Quick-Radix Sort	Partitions based on bit (depth of recursion) of the key.
Timsort	A combination of merge sort and insertion sort, originally developed for python.

4.2 Joins

Database Normalisation **Definition 4.2.1**

Structuring a relational database according to progressively stricter *normal forms* to remove redundant data, and improve integrity (prevent potential logical inconsistency under insertion/update/deletion).
See wikipedia.

4.2.1 Join Types

Normalised databases naturally require joins to re-compose data.

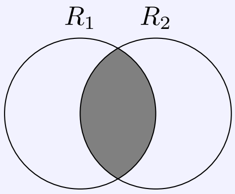
Join — **Definition 4.2.2**

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 4.2.3**

A join only returning rows from both tables which satisfy a predicate/condition.



Natural Join — **Definition 4.2.4**

Joining two tables with an implicit join clause (join on equality on a column present in both tables

$$R_1 \bowtie R_2$$

```
FROM R1 NATURAL JOIN R2
FROM R1 JOIN R2 USING(id)
```

Theta Join — **Definition 4.2.5**

Joining two tables based on a condition/predicate θ.

$$R_1 \overset{\theta}{\bowtie} R_2$$

```
FROM R1, R2 WHERE theta(R1, R2)
FROM R1 JOIN R2 ON theta(R1, R2)
```

Equi Join — **Definition 4.2.6**

A **theta join** with a single equivalence condition. A **natural join** is an implicit **equi join**.

$$R_1 \bowtie_{R_1.x = R_2.x}$$

```
FROM R1, R2 WHERE R1.x = R2.x
```

Cross Join — **Definition 4.2.7**

Just cartesian product with no selection.

$$R_1 \times R_2$$

```
FROM R1, R2
FROM R1 CROSS JOIN R2
```

Anti Join — **Definition 4.2.8**

A **theta join** using an inequality predicate

$$R_1 \bowtie_{R_1.x <> R_2.x} R_2$$

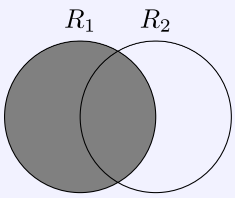
```
FROM R1 JOIN R2 ON R1.x <> R2.x
```

Outer Joins

Left 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 `NULL`).

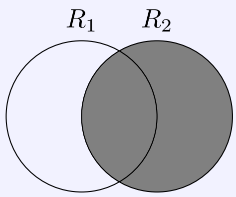



```
FROM R1 LEFT JOIN R2 ON ...
```

Right Join — **Definition 4.2.10**

$$R_1 \overset{R}{\bowtie} R_2$$

Returns all rows of R_2 even if no rows in R_1 match (in which case columns are `NULL`).

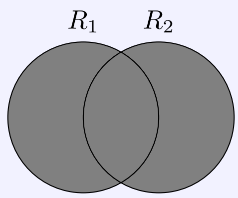



```
FROM R1 RIGHT JOIN R2 ON ...
```

Full Outer Join — **Definition 4.2.11**

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

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 4.2.1**

Which of the following are joins?

1. ```
   SELECT R.r, S.s
   FROM R, S
   WHERE R.id = S.id;
   ```
2. ```
   SELECT R.r, S.s
   FROM R, S
   WHERE R.r = R.id
   ```
3. ```
   SELECT R.r
   FROM R, S
   WHERE R.id = S.id;
   ```
4. ```
   SELECT R.r
   FROM R, S
   WHERE R.r = "some string";
   ```

---

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

4.2.2 Join Implementations

4.2.3 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...> nested_loop_join(const Table<TypesOne...> &left, const Table<TypesTwo...>
    auto result = join_empty<leftCol, rightCol>(left, right);

    for (const auto &leftElem : left.rows) {
        for (const auto &rightElem : right.rows) {
            if (get<leftCol>(leftElem) == get<rightCol>(rightElem)) {
                result.rows.emplace_back(tuple_cat(leftElem, rightElem));
            }
        }
    }

    return result;
}
```

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

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

Performance	Linear time complexity.

4.2.4 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...>
unique_sort_merge_join(const Table<TypesOne...> &leftT, const Table<TypesTwo...> &rightT) {
  auto result = join_empty<leftCol, rightCol>(leftT, rightT);

  // copy tables (so we can keep const, just reorder)
  auto left = leftT;
  auto right = rightT;

  sort(left.rows.begin(), left.rows.end(), [](auto const &a, auto const &b) {
    return get<leftCol>(a) < get<leftCol>(b);
  });
  sort(right.rows.begin(), right.rows.end(), [](auto const &a, auto const &b) {
    return get<rightCol>(a) < get<rightCol>(b);
  });

  auto leftIndex = 0;
  auto rightIndex = 0;

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

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

  return result;
```

Non-Unique *Extra Fun!* **4.2.1**

A sort-merge that does not require uniqueness is included in this chapter's code directory.

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

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.

4.2.5 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`
- We assume each value in a table is unique, otherwise a `unordered_multimap` is required.

```
template <size_t leftCol, size_t rightCol, typename... TypesOne, typename... TypesTwo>
Table<TypesOne..., TypesTwo...> unique_hash_join(const Table<TypesOne...> &left, const Table<TypesTwo...>
  auto result = join_empty<leftCol, rightCol>(left, right);

  using leftColType = typename tuple_element<leftCol, tuple<TypesOne...>>::type;

  // we should ideally choose the smallest table here -> smallest hashmap
  unordered_map<leftColType, const tuple<TypesOne...> *> leftContents(left.rows.size());

  for (const tuple<TypesOne...> &elem : left.rows) {
    leftContents.insert(make_pair(get<leftCol>(elem), &elem));
  }

  for (auto &elem : right.rows) {
    if (leftContents.contains(get<rightCol>(elem))) {
      result.rows.emplace_back(tuple_cat(*leftContents[get<rightCol>(elem)], elem));
    }
  }

  return result;
}
```

$$\begin{aligned}&\Theta(|build| + |probe|)\text{best case}\\ &O(|build| \times |probe|)\text{worst case}\end{aligned}$$

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

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

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

Bucket Based Hashmaps — ***Extra Fun!* 4.2.2**

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 Hashing

A hash function is used to lookup keys in a hashmap.

```
template<typename K> using Hasher = size_t (const K&);
```

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

Typical Hashers — ***Extra Fun!* 4.3.1**

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

Hashes can collide, and hence we need a way to resolve this.

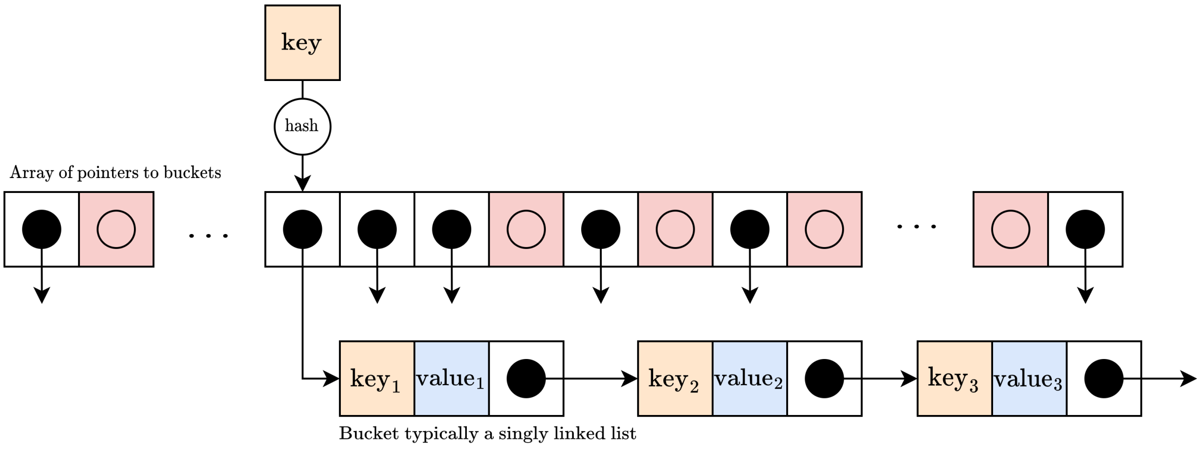
Hash it out — **Example Question 4.3.1**

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(const int& data) {
  return static_cast<size_t>(data) % MODULUS;
}
```

4.3.2 Bucket Hashmap (Separate Chaining)

Collisions are resolved using linked-list buckets.



Simple Easy to implement (no extra logic for collisions required, can just insert/delete from linked list buckets).

Locality Linked list nodes are usually allocated separately, resulting in a random access pattern on larger buckets.

```
template<typename K, typename V, Hasher<K> hasher>
class Buckets {

    struct BucketNode {
        K key;
        V value;
        BucketNode* next;
    };

    std::vector<BucketNode*> _buckets;
    size_t _size;

    const size_t _get_index(const K& key) const noexcept {
        return hasher(key) % _buckets.size();
    }

    bool _resize_policy() const noexcept {
        // if on average, more than two per bucket - resize.
        // is this a good policy? ==> depends on (hasher, cost of resize)
        return _size / _buckets.size() > 2;
    }

    void _resize() noexcept {
        vector<BucketNode*> old_buckets(_buckets.size() * 2);
        std::swap(old_buckets, _buckets);
        for (auto& bucket : old_buckets) {
            for (auto curr = bucket; curr;) {
                BucketNode*& dest = _buckets.at(_get_index(curr->key));
                BucketNode* next = curr->next;
                curr->next = dest;
                dest = curr;
                curr = next;
            }
        }
    }

public:
```

```cpp
// INV: initial_buckets > 0
Buckets(size_t initial_buckets = 16) : _buckets(initial_buckets), _size(0) {}

bool insert(K key, V val) {
    if (_resize_policy()) _resize();
    BucketNode*& bucket = _buckets.at(_get_index(key));
    for (const BucketNode* b = bucket; b; b = b->next) {
        if (b->key == key) return false;
    }
    bucket = new BucketNode{ .key=std::move(key), .value=std::move(val), .next=bucket };
    _size++;
    return true;
}

V* find(const K& key) noexcept {
    for (BucketNode* b = _buckets.at(_get_index(key)); b; b = b->next) {
        if (b->key == key) return &b->value;
    }
    return nullptr;
}

bool contains(const K& key) const noexcept {
    for (const BucketNode* b = _buckets.at(_get_index(key)); b; b = b->next) {
        if (b->key == key) return true;
    }
    return false;
}

bool erase(const K& key) noexcept {
    BucketNode*& first_node = _buckets.at(_get_index(key));

    if (first_node) {
        if (first_node->key == key) {
            BucketNode* next_node = first_node->next;
            delete first_node;
            first_node = next_node;
            _size--;
            return true;
        } else {
            BucketNode* prev = first_node;
            for (BucketNode* curr = prev->next; curr; prev = curr, curr = curr->next) {
                if (curr->key == key) {
                    prev->next = curr->next;
                    delete curr;
                    _size--;
                    return true;
                }
            }
            return false;
        }
    } else {
        return false;
    }
}

size_t size() const noexcept { return _size; }

friend std::ostream &operator<<(std::ostream &os, const Buckets<K, V, hasher> & ht) {
    os << "Hash Table: " << type<Buckets<K, V, hasher>>() << endl;
    os << "Size: " << ht.size() << endl;
```

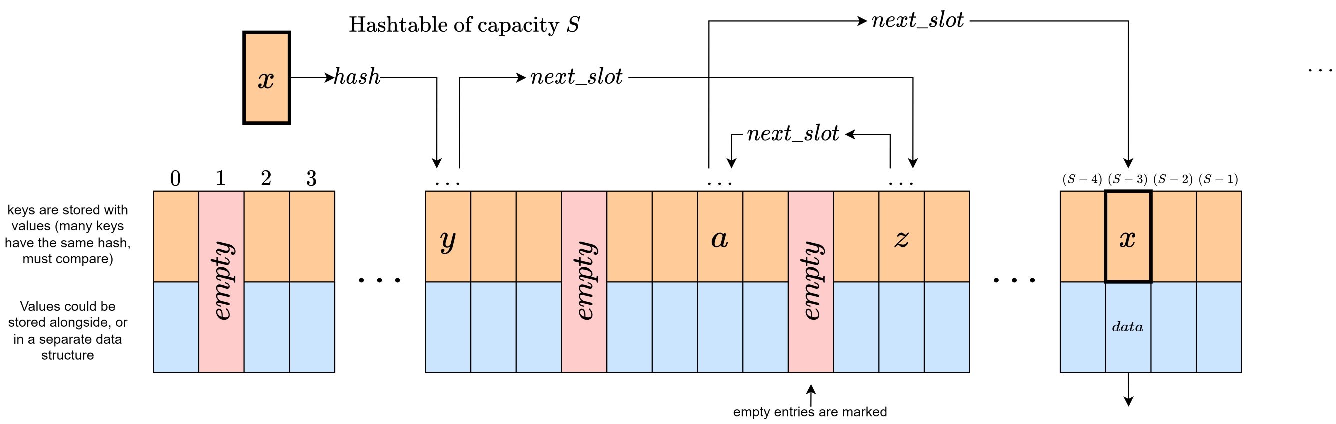
```
        os << "Buckets: " << ht._buckets.size() << endl;
        for (size_t i = 0; i < ht._buckets.size(); i++) {
            BucketNode* first_node = ht._buckets[i];
            os << i << ": ";
            if (first_node) {
                for (BucketNode* b = first_node; b; b = b->next) {
                    os << "-> " << "{" << b->key << ":" << b->value << "}";
                }
            } else {
                os << "<empty>";
            }
            os << endl;
        }
        return os;
    }

    static_assert(IsHashMap<Buckets, K, V>, "BucketMap is not a hashmap!");
};
```

Other Structures — ***Extra Fun!* 4.3.2**

Balanced Tree	If keys are ordered, then a self-balancing tree can provide $\log(n)$ lookup, potentially better then a bucket based hashmap under many collisions (bucket based hashmap can deteriorate into iterating over large lists).
Dynamic Perfect Hashing	Each bucket is a second hashtable. Requires a perfect hash over k^2 slots (where the hashtables have k slots).

4.3.3 Probing Hashmap (Open Addressing)

Collisions are resolved by retrying on a new slot (determined by a probe).



We can define an interface for a probe using a concept.

```
template<typename State>
struct ProbeResult {
    size_t hash;
    State state;
};

template<typename K, typename Probe>
concept IsProbe = requires {
    {
        Probe::initial(std::declval<const K&>())
    } noexcept -> std::same_as<ProbeResult<typename Probe::State>>;
    {
        Probe::collision(std::declval<const ProbeResult<typename Probe::State>&>())
```

```
    } noexcept -> std::same_as<ProbeResult<typename Probe::State>>;
};
```

We can then implement different probes, ideally with the following qualities:

High Locality	When detecting a conflict, the real key is close/same page/line.
Not too High!	Very high locality will result in parts of the hash table being saturated, and long probe chains.
No Holes	We want to avoid leaving holes/wasted memory (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 (here we assume the hashmap structure modulates our output).

```
template<typename K, Hasher<K> hasher, size_t DISTANCE>
struct Linear {
    struct State {}; // no state required for linear hashing

    static ProbeResult<State> initial(const K& key) noexcept {
        return { .hash=hasher(key), .state={} };
    }

    static ProbeResult<State> collision(const ProbeResult<State>& prev) noexcept {
        return { .hash=prev.hash+DISTANCE, .state={} };
    }
};
```

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, Hasher<K> hasher>
struct Quadratic {
    struct State {
        size_t original_hash;
        size_t collisions = 0;
    };

    static ProbeResult<State> initial(const K& key) noexcept {
        size_t hash = hasher(key);
        return {
            .hash=hash,
            .state={ .original_hash=hash, .collisions=0 }
        };
    }
    static ProbeResult<State> collision(const ProbeResult<State>& prev) noexcept {
        size_t collisions = prev.state.collisions + 1;
        return {
            .hash=prev.state.original_hash + collisions * collisions,
            .state={
```

```
                .original_hash=prev.state.original_hash,
                .collisions=collisions
            }
        };
    }
};
```

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

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

Rehashing

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

```
template<typename K, Hasher<K> hasher, Hasher<size_t> reHasher>
struct ReHash {
    struct State {};

    static ProbeResult<State> initial(const K& key) noexcept {
        return ProbeResult{ .hash=hasher(key), .state={} };
    }

    static ProbeResult<State> collision(const ProbeResult<State>& prev) noexcept {
        return { .hash=reHasher(prev.hash), .state={} };
    }
};
```

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).

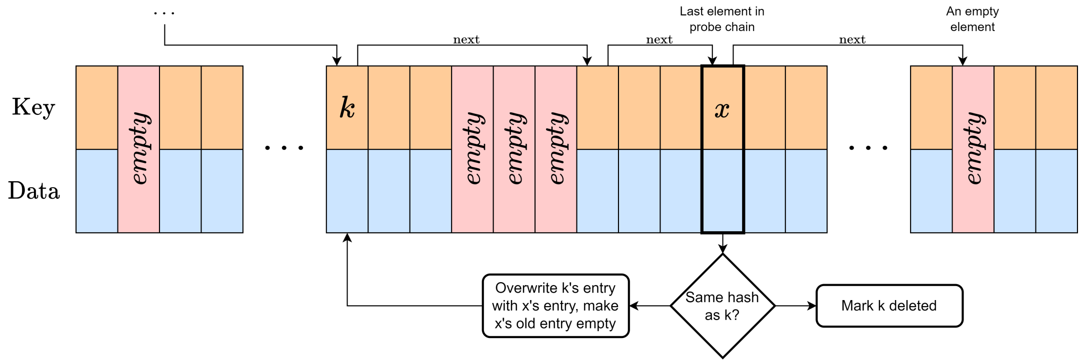
Resizing

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

- 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



Probing Hashmap Implementation

```
  struct Entry {
    K key;
    V val;
  };
  struct Deleted {
    K key;
  };
  struct Empty {};

  // entries are default constructed as Empty
  std::vector<std::variant<Empty, Deleted, Entry>> _table;
  size_t _size;

  bool _resize_policy() const noexcept { return _size > _table.size() / 2; }

  void _resize() noexcept {
    // we know the old table has unique entries, and can take advantage of this
    std::vector<std::variant<Empty, Deleted, Entry>> _old_table(_table.size() *
                                                                2);
    std::swap(_old_table, _table);
    for (auto &slot : _old_table) {
      if (holds_alternative<Entry>(slot)) {
        Entry &entry = std::get<Entry>(slot);
        for (auto probe = Probe::initial(entry.key);;
             probe = Probe::collision(probe)) {
          auto &slot = _table.at(probe.hash % _table.size());
          if (holds_alternative<Empty>(slot)) {
            slot = std::move(entry);
            break;
          }
        }
      }
    }
  }

public:
  // INV: initial_buckets > 0
  Probing(size_t initial_capacity = 16) : _table(initial_capacity), _size(0) {}

  bool insert(K key, V val) {
    if (_resize_policy())
      _resize();

    for (auto probe = Probe::initial(key);; probe = Probe::collision(probe)) {
```

```
    auto &slot = _table.at(probe.hash % _table.size());

    if (std::holds_alternative<Empty>(slot) ||
        std::holds_alternative<Deleted>(slot)) {
      slot = Entry{.key = std::move(key), .val = std::move(val)};
      _size++;
      return true;
    } else if (std::get<Entry>(slot).key == key) {
      // key is already present
      return false;
    }
  }
}
V *find(const K &key) noexcept {
  for (auto probe = Probe::initial(key);; probe = Probe::collision(probe)) {
    auto &slot = _table.at(probe.hash % _table.size());
    if (std::holds_alternative<Empty>(slot)) {
      // end of probe chain,
      return nullptr;
    } else if (std::holds_alternative<Deleted>(slot)) {
      Deleted &deleted = std::get<Deleted>(slot);
      if (deleted.key == key) {
        // key was deleted, if reinserted, it would be here or earlier in the
        // probe chain.
        return nullptr;
      }
    } else if (std::holds_alternative<Entry>(slot)) {
      // a key is present, check the key
      Entry &entry = std::get<Entry>(slot);
      if (entry.key == key) {
        return &entry.val;
      }
    }
  }
}
bool contains(const K &key) const noexcept {
  for (auto probe = Probe::initial(key);; probe = Probe::collision(probe)) {
    const auto &slot = _table.at(probe.hash % _table.size());
    if (std::holds_alternative<Empty>(slot)) {
      // end of probe chain,
      return false;
    } else if (std::holds_alternative<Deleted>(slot)) {
      const Deleted &deleted = std::get<Deleted>(slot);
      if (deleted.key == key) {
        // key was deleted, if reinserted, it would be here or earlier in the
        // probe chain.
        return false;
      }
    } else if (std::holds_alternative<Entry>(slot)) {
      // a key is present, check the key
      const Entry &entry = std::get<Entry>(slot);
      if (entry.key == key) {
        return true;
      }
    }
  }
}
bool erase(const K &key) noexcept {
  for (auto probe = Probe::initial(key);; probe = Probe::collision(probe)) {
    auto &slot = _table.at(probe.hash % _table.size());
```

```
      if (std::holds_alternative<Empty>(slot)) {
        // end of probe chain,
        return false;
      } else if (std::holds_alternative<Deleted>(slot)) {
        Deleted &deleted = std::get<Deleted>(slot);
        if (deleted.key == key) {
          // key was deleted, if reinserted, it would be here or earlier in the
          // probe chain.
          return false;
        }
      } else if (std::holds_alternative<Entry>(slot)) {
        // a key is present, check the key
        Entry &entry = std::get<Entry>(slot);
        if (entry.key == key) {
          slot = Deleted{.key = std::move(entry.key)};
          _size--;
          return true;
        }
      }
    }
  }
  size_t size() const noexcept { return _size; }

  friend std::ostream &operator<<(std::ostream &os,
                                  const Probing<K, V, Probe> &ht) {
    os << "Hash Table: " << type<Probing<K, V, Probe>>() << endl;
    os << "Size: " << ht.size() << endl;
    os << "Slots: " << ht._table.size() << endl;
    for (size_t i = 0; i < ht._table.size(); i++) {
      const auto &slot = ht._table.at(i);
      os << i << ": ";
      std::visit(overloaded{
                     [&](const Entry &entry) {
                       os << "-> {" << entry.key << " : " << entry.val << "}";
                     },
                     [&](const Deleted &deleted) {
                       os << "<Deleted " << deleted.key << ">";
                     },
                     [&](const Empty &entry) { os << "<Empty>"; },
                 },
                 slot);
      os << endl;
    }
    return os;
  }
```

4.3.4 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.

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

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

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

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

- Increase the page size (huge pages).

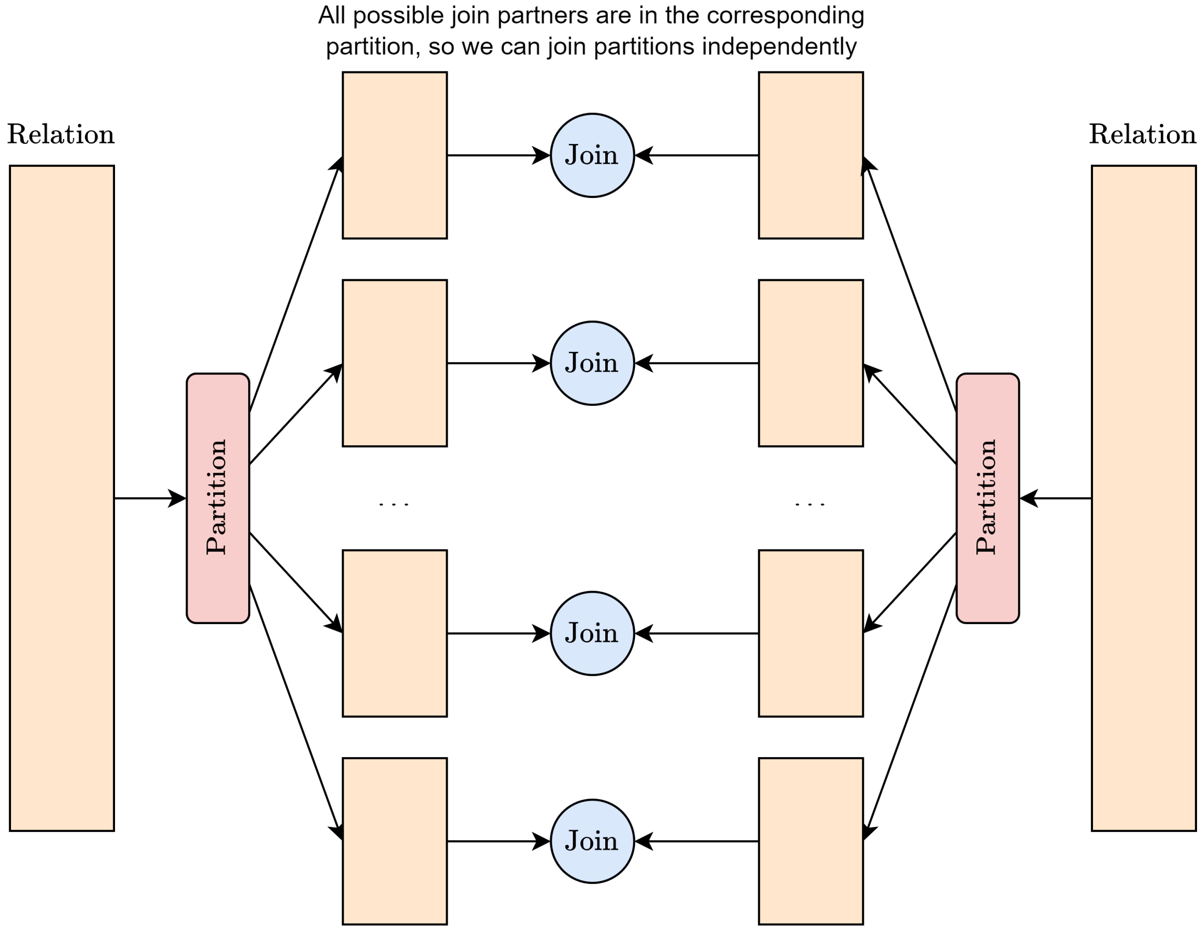
- Make the access pattern *more* sequential.

The access pattern for hashtables is typically random (often intentional by hash function).

Hashtable Thrashing **Definition 4.3.1**

Thrashing due to random access pattern associated with hashtable access, on a large hashtable.

- e.g high page IO when hash-joining two large relations, as the hashtable is too large to fit in the buffer pool.



- As each partition can be joined independently, we can do one at a time, and hence each partition's hash-join's hashtable can fit within the buffer pool.
- Alternatively, we could join all in parallel.

The partitioning function can be as simple as $key \mod No.Partitions$.

4.3.5 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 4.3.2**

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 4.3.3**

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.

The elephant in the room — ***Extra Fun!* 4.3.3**

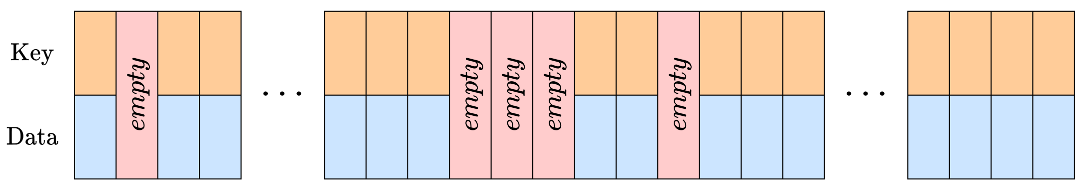
Among other DBMS, Postgres supports many types of index (documentation here)

```
/* 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');
```

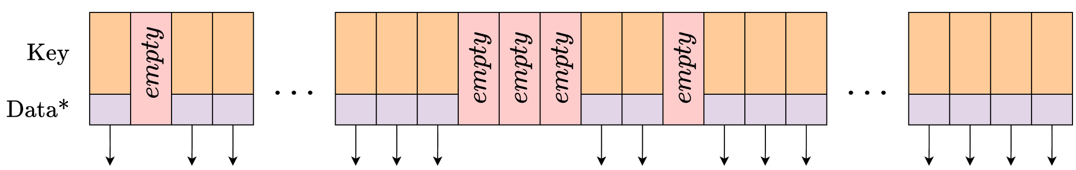
4.3.6 Hash Indexes

An index backed by a hash table.

Clustered Hash Index



Unclustered Hash Index



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:

Hash-Joins & Aggregation Perform well and remove build phase (provided they index on the columns joining).

Equality Selection Can reduce number of candidate columns if not all columns are indexed

```
SELECT * FROM table_name WHERE column1 = some_value;
```

Limited Applicability Not useful for queries not using equality.

4.3.7 Bitmap Indexing

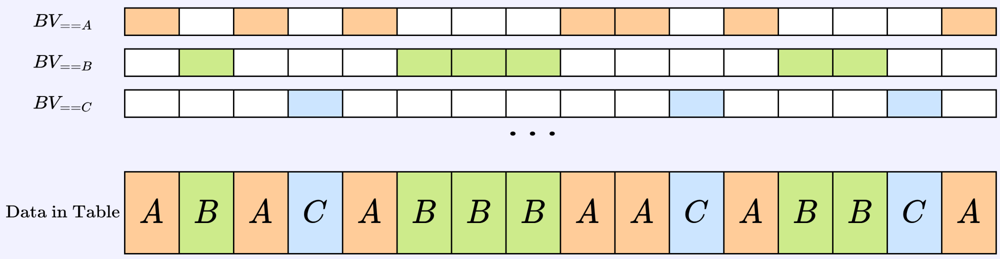
Bit Vector **Definition 4.3.4**

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 4.3.5**



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)

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

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 <cstddef>
#include <cstdint>
#include <iostream>
#include <vector>

using namespace std;
```

```cpp
// 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 $\to$ 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 are 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 $\to$ 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 ame 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 $x < x < y$, what proportion of the 1s in the bitvector are not for the value z.

Equi-Width — **Definition 4.3.6**

$$width = \frac{max(column) - min(column)}{number\_bins}$$

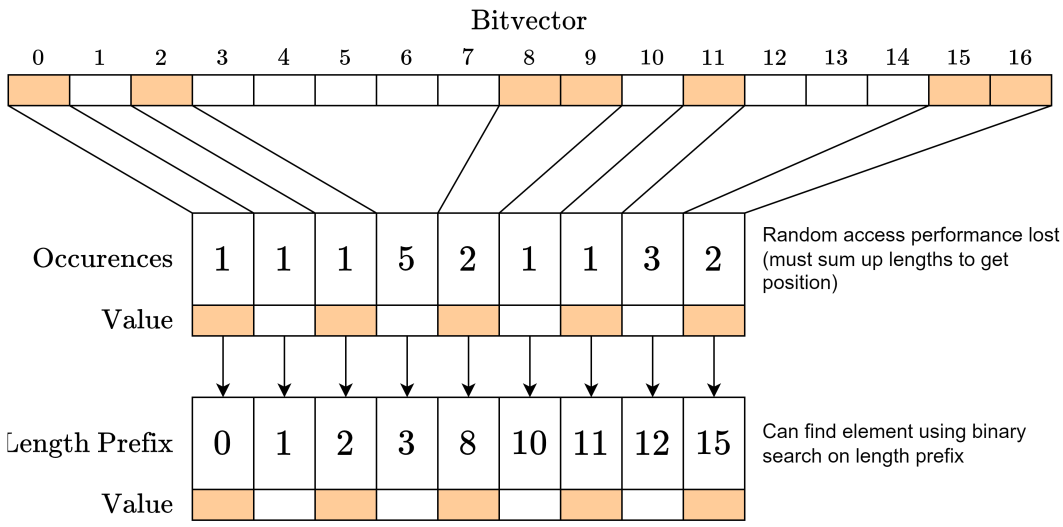
Split range into several equal size bins. Useful for uniformly distributed (or near) data.

Height Binning — **Definition 4.3.7**

All bins should contain the same number of values.

- Construction difficult (usually sort, determine quatiles on a sample)
- False-positive rate is value independent
- As table changes, may need to re-bin.

Run Length Encoding



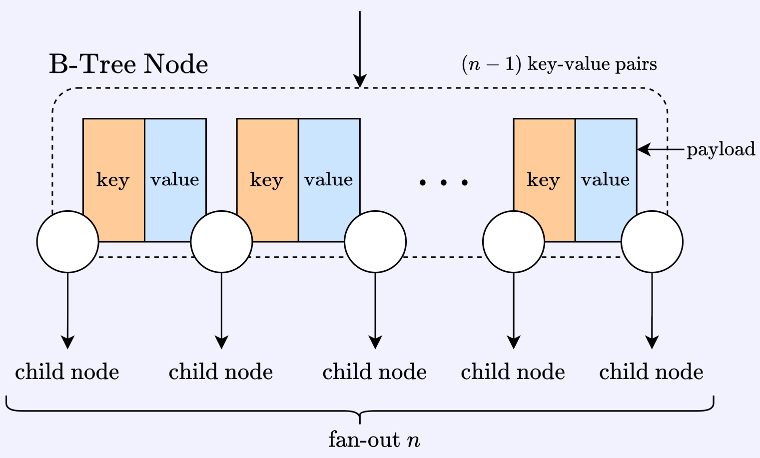
4.3.8 B-Trees

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 $\rightarrow$ many page faults occur to fetch only a few nodes). Databases are I/O bound (here the I/O is page faults).

B-Tree **Definition 4.3.8**



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 $O(\log n)$

4.3.9 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!!!

4.3.10 Foreign Key Indices

Most joins are using a foreign key relation.

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

Cheap	Use a small amount of space, only need to add an extra pointer when updating, no extra optimisation effort needed (other than *if an fk-index exists, use it*).
Used	Foreign key constraints usually are only created where users intend to join.

Chapter 5

Processing Models

5.1 Motivation

Processing Model **Definition 5.1.1**

A mechanism used to connect operators acting on data in a query.

- Choice is critical to database design.

Function Objects **Definition 5.1.2**

References to code that can be passed, invoked, change state and produce values. See these notes associated code for more detail.

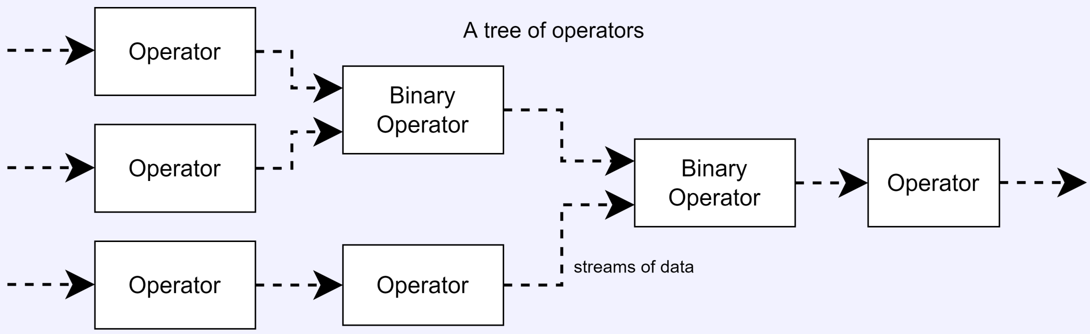
```
#include <functional>

std::function<int(int, int)> add = [ /* captures */ ](int a, int b) { return a + b; }
```

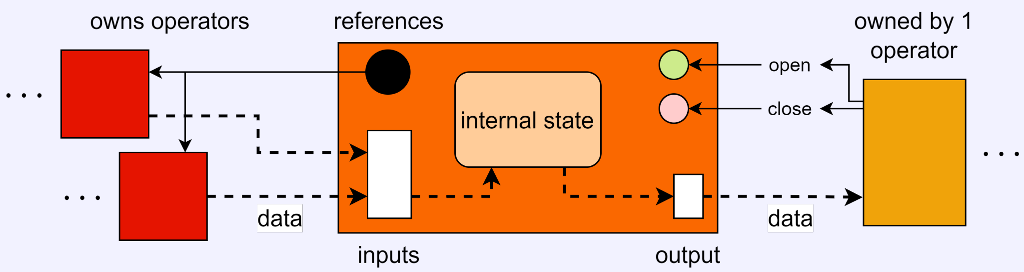
See C++11 Lambdas

- Can capture variables (value and references to) (also called closures).
- Used to implement single abstract method classes in some languages (e.g kotlin, java)

5.2 Volcano Processing

Volcano Processing Model **Definition 5.2.1**





Data is fed chunk by chunk (row) through a tree of operators.

- Older design (influential in the 80s) with a focus on design practices over performance. At the time this was an alternative to ad-hoc implementation.
- Uses non-relational physical algebra (specialized to be useful in expressing queries for a physical plan, rather than as an abstraction for the programmer).

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! Function calls are expensive, virtual calls even more so! Operators work my virtual calls to on parent operators per tuple, so the number of calls grows with the table size.

5.2.1 Operators

A basic interface for operators can be devised as:

```
template <typename T> struct Operator {
  virtual void open() = 0;
  virtual void close() = 0;
  virtual std::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.

But why not RAII	***Extra Fun!* 5.2.1**

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 = std::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(std::shared_ptr<TableType> t) : _table(t), _index(0) { assert(_table); }

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

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

Project

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

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

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

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

Select

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

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

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

  std::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;
  }
```

Union

```cpp
template <typename T> struct Union : Operator<T> {
  Union(std::unique_ptr<Operator<T>> leftChild,
        std::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();
  }

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

Difference

Pipeline Breaker — **Definition 5.2.2**

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.

```cpp
template <typename T> struct Difference : Operator<T> {
  Difference(std::unique_ptr<Operator<T>> fromChild,
             std::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();
  }

  std::optional<T> next() override {
    auto candidate = _fromChild->next();
    // keep gettihg 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:
  std::unique_ptr<Operator<T>> _fromChild, _subChild;
  std::unordered_set<T> _subBuffer;
};
```

Cartesian/Cross Product

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

```cpp
template <typename A, typename B>
struct BreakingCrossProduct : Operator<std::tuple<A, B>> {
  BreakingCrossProduct(std::unique_ptr<Operator<A>> leftChild,
                       std::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();
  }

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

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

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

private:
  std::unique_ptr<Operator<A>> _leftChild;
  std::unique_ptr<Operator<B>> _rightChild;
  std::optional<A> _leftCurrent;
  size_t _rightIndex;
  std::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<std::tuple<A, B>> {
  CrossProduct(std::unique_ptr<Operator<A>> leftChild,
               std::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();
  }
```

```
  std::optional<std::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 had 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 std::make_tuple(_leftCurrent.value(),
                             _rightBuffered[_rightOffset++]);
    } else {
      return {};
    }
  }

private:
  std::unique_ptr<Operator<A>> _leftChild;
  std::unique_ptr<Operator<B>> _rightChild;
  std::optional<A> _leftCurrent;
  std::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 = std::function<S(std::optional<S>, T)>;
  using Grouping = std::function<K(T)>;

  GroupBy(std::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();

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

    _hashTable = std::vector<std::optional<std::pair<K, S>>>(
        childValues.size(), std::optional<std::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
                          : std::optional<S>();
      _hashTable[slot] = std::optional<std::pair<K, S>>(
          std::make_pair<K, S>(move(key), _aggregation(prev_val, val)));
    }

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

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

  std::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;
  std::unique_ptr<Operator<T>> _child;
  std::vector<std::optional<std::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;
```

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

auto scan = std::make_unique<Scan<Row>>(data);

// Group by for single column
auto groupBySecondCol = [](Row r) { return r[1]; };
auto aggregateSecondCol = [](std::optional<Row> r1, Row r2) {
  if (r1.has_value()) {
    return Row{std::max(std::get<int>(r1.value()[0]), std::get<int>(r2[0])),
               r2[1]};
  } else {
    return Row{r2[0], r2[1]};
  }
};

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

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

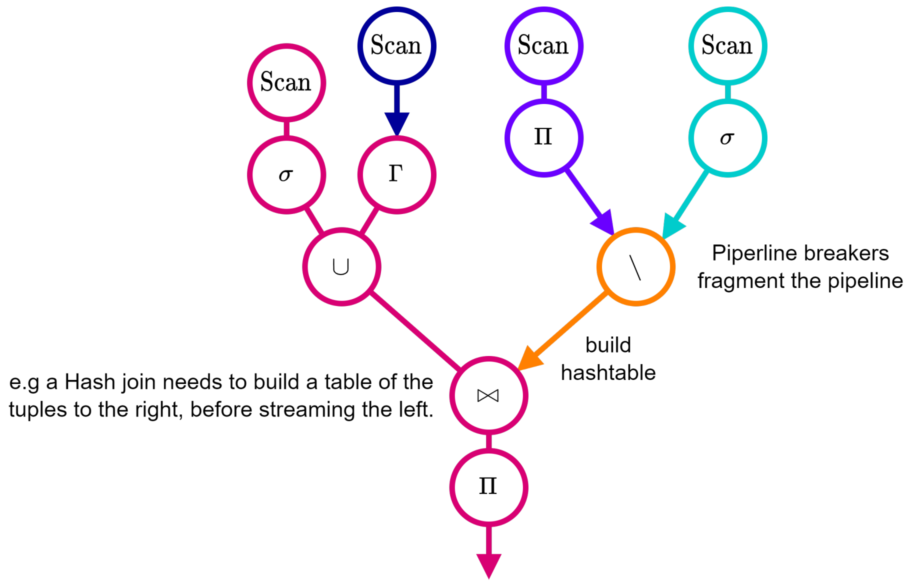
```
[ 3 e]
[ 5 c]
[ 1 d]
```

Run it! ***Extra Fun!* 5.2.2**

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

5.2.2 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.

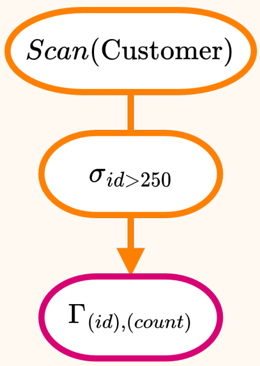
- If all buffers in a fragment fit in memory, there is no I/O
- Otherwise: sequential access → number of occupied pages, random access/out of order → one page I/O per access (an upper bound & almost certainly an over estimate)

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

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

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

Basic GroupBy **Example Question 5.2.1**




```
CREATE TABLE Customer (
  id      i32,
  name    STRING, -- Key into compressed dictionary
  address STRING, -- Key into compressed dictionary
  nation  i32,
  phone   i32,
  accNum  i32
);
```

- Customer has 10,000 tuples
- Strings are represented by a 32 bit integer key into a compressed dictionary.
- $\sigma_{id>250}$ has 30% selectivity.
- $\Gamma_{(id),(count)}$ has grouping cardinality of 9
- Page size is 64 B
- Buffer pool is 512 KB

---

1. $Scan(Customer)$:

$$size(Customer) = (6 \times 4) \times 10,000 = 240,000 \Rightarrow pages(Customer) = \left\lceil \frac{240,000}{64} \right\rceil = 3,750$$

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

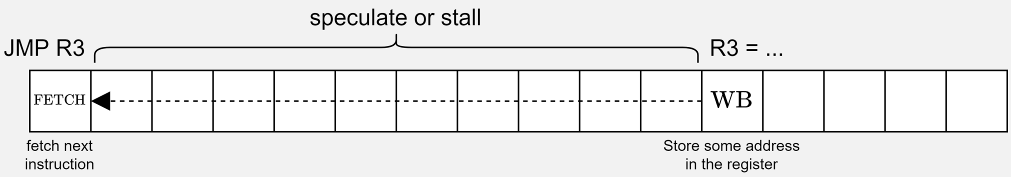
2. $\sigma_{id<250}(\dots)$
   - Not a pipeline breaker, so no need to buffer.
   - No IO costs as child *Scan* operation passes tuple in memory.
3. $\Gamma_{(id),(count)}$
   For the grouping we assume a hashtable overallocation factor of 2, the table contains *count* and grouping attribute (id).

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

CPU Efficiency

Not all function calls are equal. ***Extra Fun!* 5.2.3**

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:

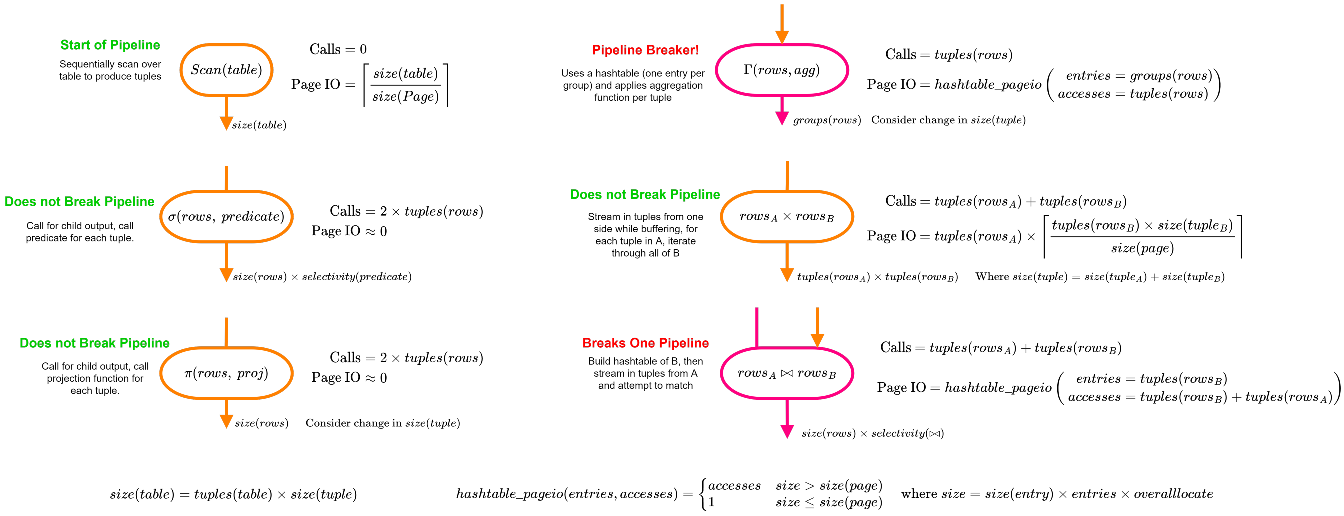
- Jump to an immediate value (typically pc-relative immediate offset in the jump instruction), as the jump location is part of the instruction, there is no hazard. But the function to jump to must be known at compile time. Still affects returns (jump to link register/return address register) (though this should be very fast due to return-address stack branch predictors).
- Determine the function to call at compile time (jump to label in asm $\rightarrow$ jump to immediate pc-relative address). This is still costly (depending on calling convention), so we can go further an inline.
- 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/σ	2	Call to read input, call to apply predicate.
Project/Π	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.

5.2.3 Operations Calculations

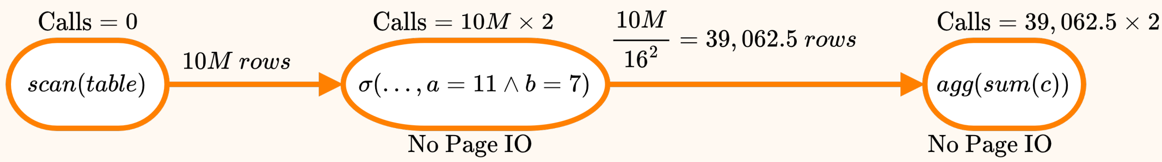
Spanned Pages - Volcano Processing



Selective — **Example Question 5.2.2**

```sql
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.



$$\text{Page IO} = \left\lceil \frac{10M \times (3 \times 4)}{64} \right\rceil = 1,875,000 \qquad \text{Total Function Calls} = 20078125$$

5.3 Bulk Processing

Bulk Processing — **Definition 5.3.1**

Queries are processed in batches.

- Turn *control dependencies* to *data dependencies*.
- Apply operator to a buffer of tuples, copy or pass references to buffers between operators.
- Reduces the number of function calls (e.g 1 per tuple per operator → 1 per operator).

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 5.3.1**

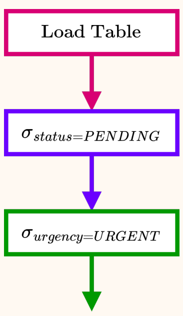
Translate the following to use the `select_eq` implementation above.

```
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 5.3.2**



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

- Selectivity of both σ 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 $187,500 + 46,875 + 11,719 = 246094$ IO actions.

5.3.1 By-Reference Bulk Processing

By-Reference Bulk Processing **Definition 5.3.2**

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:

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

5.3.2 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 5.3.3**

UNFINISHED!!!

Chapter 6

Optimisation

6.1 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.

6.1.1 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!* 6.1.1**

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.

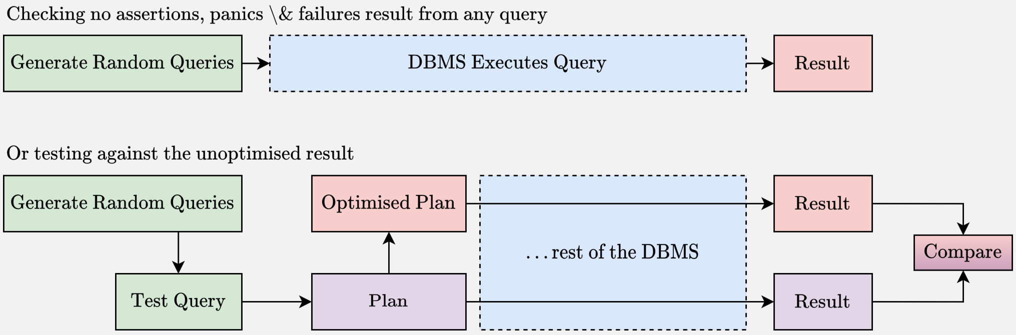
```
g++ -fprofile-generate myprog.cpp # Compile instrumented version
./myprog.cpp                      # Generates myprog.gcda
g++ -fprofile-use myprog.cpp      # Use profile when optimising
```

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).

Fuzzing *Extra Fun!* **6.1.2**

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).

6.1.2 Query Equivalence

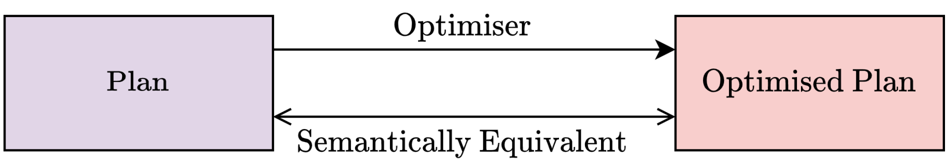
Semantic Equivalence **Definition 6.1.1**

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

Closure (Mathematics) **Definition 6.1.2**

(*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!* **6.1.3**

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

6.2 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.

WACC Peephole — *Extra Fun!* 6.2.1

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

```
mov r1, r1 ; redundant move
```

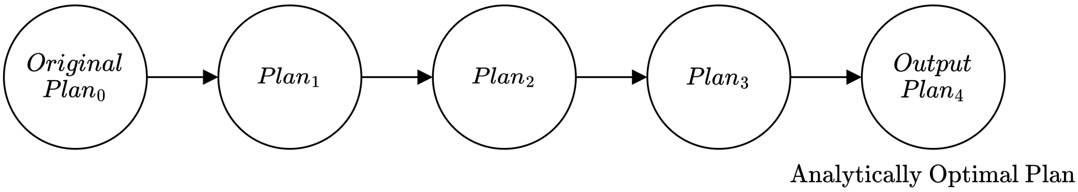
```
str r4, [sp, #8] ; overwritten store
str l3, [sp, #8]
```

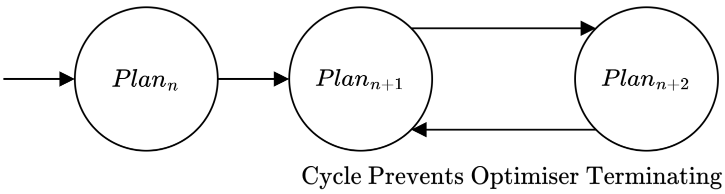
- Need some order with which to traverse the plan
- Need a set of patterns/rules to apply.

6.2.1 Avoiding Cycles

Analytically Optimal Plan — Definition 6.2.1

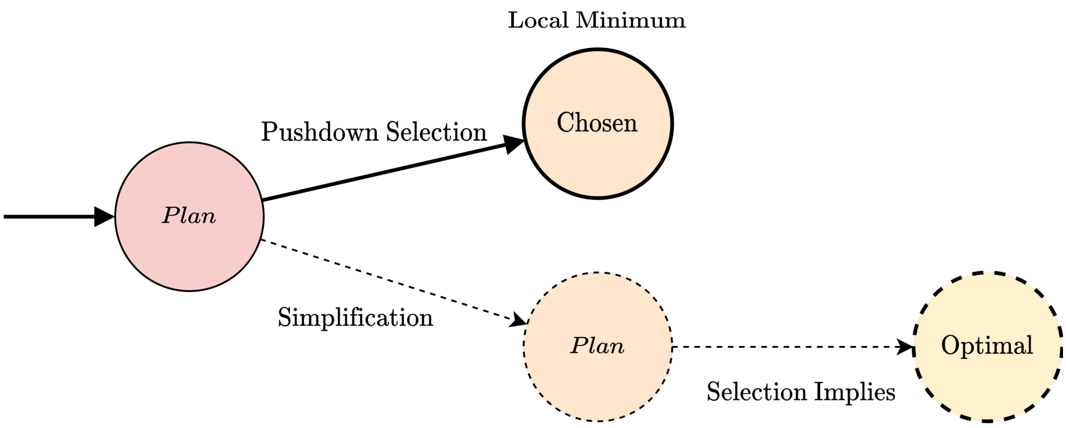
The final plan output of the optimiser (not necessarily the most optimal plan).





Avoiding this requires careful rule selection.

6.2.2 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.

Loops	Developer must be careful to not introduce potential loops in rule application.
Local Optima	Typically many choices of rule to apply ⇒ local optima.

6.3 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*
Data	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*.

Logical	*Algorithm-Agnostic*	Deals only with relational algebra.
Physical	*Algorithm-Aware*	Can use different operator implementations, indices etc.
Rule-Based	*Data-Agnostic*	Applying optimisation rules that are almost always beneficial.
Cost-Based	*Data-Aware*	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).

6.4 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.

Your turn!	***Extra Fun!* 6.4.1**
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!	

6.4.1 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:

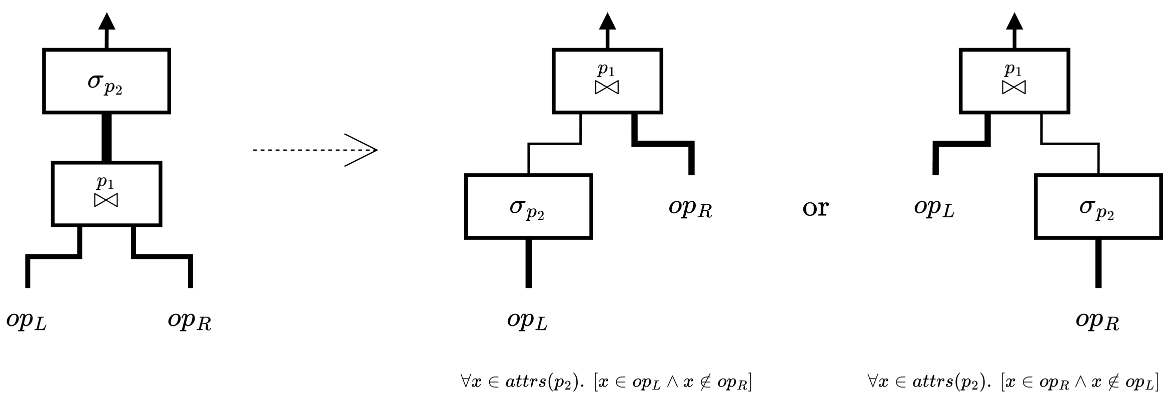
- Higher cardinality (more tuples) $\Rightarrow$ Higher Cost
- Joins usually increase cardinality, or leave unchanged
- Selections reduce cardinality
- Aggregations reduce cardinality
- Data access is more expensive than function evaluation (can assume generally, without exposing operator implementation)

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 attributed from one side of the join.

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

```sql
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)
```

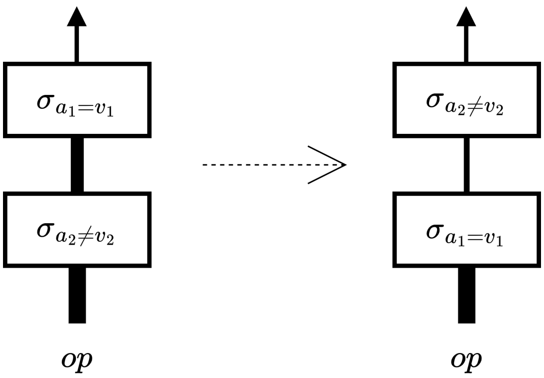
Dont push me down! — **Example Question 6.4.1**

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 >=) < <>.

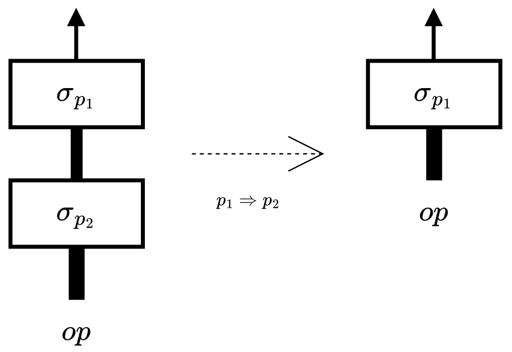
```
SELECT * FROM ( SELECT * FROM op WHERE a2 <> v2 ) WHERE a1 == v1;
```

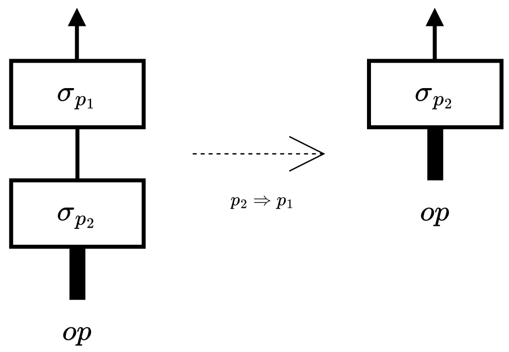
. . . is optimised to . . .

```
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.

6.4.2 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 know.

Uniform Distribution

If we assume all values are equally likely:

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

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

$$selectivity(a = v) \triangleq P(a = v) \equiv \frac{histogram_a.v}{histogram_a.total}$$

- 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:

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

Binned Histograms — ***Extra Fun!* 6.4.2**

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

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

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

6.5 Physical Optimisation

Physical Plan — **Definition 6.5.1**

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*)

6.5.1 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 `ORDER BY`, 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.

6.5.2 Cost Based Physical Optimisation

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

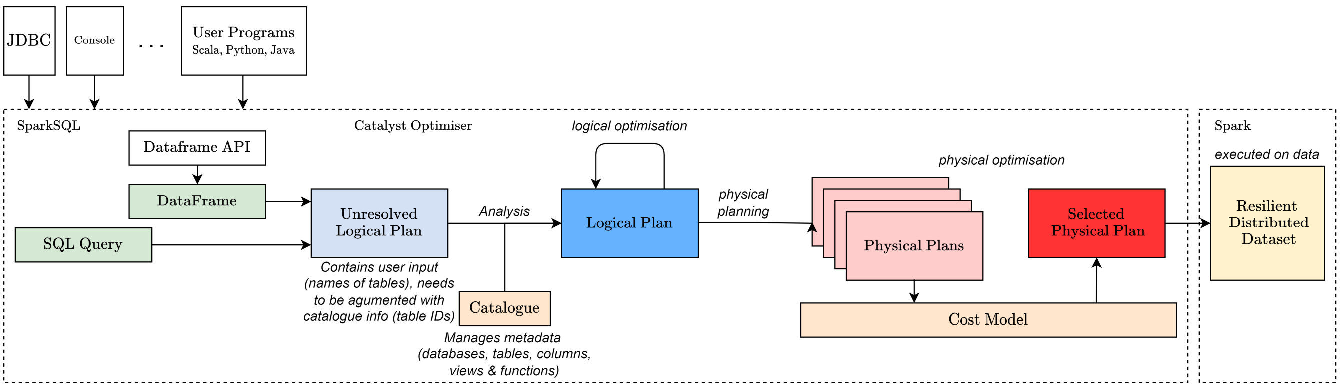
Data	Consider cardinalities & how this affect operator choice (e.g choose sort-merge join over hash if the required hashtable is too large for the buffer pool).
Hardware	Function call overhead (for this architecture), buffer pool size, access latencies, available parallelism (hardware threads).
Algorithm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

This is the current state of the art in optimisation.

6.6 SparkSQL

SparkSQL Logical Optimiser — ***Extra Fun!* 6.6.1**

You can find the source for SparkSQL's catalyst optimiser's logical optimisations here on github.



- *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 7

Transactions

40007 - Introduction to Databases — ***Extra Fun!* 7.0.1**

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

7.1 SQL Transaction

```sql
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 7.1.1**

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 7.1.1**

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

```sql
BEGIN TRANSACTION T1                     BEGIN TRANSACTION T2
INSERT INTRO table1 VALUES (1,9);
                                         INSERT INTRO table1 VALUES (17,90);
SELECT sum(column1) FROM table1;
                                         SELECT sum(column1) FROM table1;
END TRANSACTION                          END TRANSACTION
```

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

7.1.1 ACID Properties

Atomicity — **Definition 7.1.2**

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

Consistency — **Definition 7.1.3**

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 7.1.4**

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 7.1.5**

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).

7.2 Histories

Read/Write Locks — **Definition 7.2.1**

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 $first \prec second$.

7.3 Anomalies

Dirty Read / Read After Write / Uncommitted Dependency **Definition 7.3.1**

A transaction reads uncommitted data.

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

```
BEGIN TRANSACTION T1                          BEGIN TRANSACTION T2
                                              UPDATE table SET a = 5 WHERE b = 1;
SELECT a FROM table WHERE b = 1;
END TRANSACTION
-- committed T1 depends on uncommitted T2
                                              END TRANSACTION
```

Non-Repeatable Read **Definition 7.3.2**

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 7.3.3**

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 7.3.4**

A transaction overwrites uncommitted data from another transaction.

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

```
BEGIN TRANSACTION T1                          BEGIN TRANSACTION T2
                                              UPDATE table SET  a = 5 WHERE  b = 1;
UPDATE table SET  a = 9 WHERE  b = 1;
UPDATE table SET  a = 9 WHERE  b = 3;
                                              UPDATE table SET  a = 5 WHERE  b = 3;
END TRANSACTION                               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 7.3.5**

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

```
BEGIN TRANSACTION T1                                BEGIN TRANSACTION T2
-- read a                                           -- read c
                                                    UPDATE table SET a = c WHERE b = 1;
UPDATE table SET c = a WHERE b = 1;
END TRANSACTION                                     END TRANSACTION
-- a <> c (a and b has been swapped)
```

Inconsistent Analysis — **Definition 7.3.6**

A transactions 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                                BEGIN TRANSACTION T2
                                                    UPDATE table SET a = 9 WHERE  b = 1;
SELECT sum(a) FROM table;
                                                    UPDATE table SET a = 17 WHERE  b = 3;
SELECT sum(a) FROM table;
-- sum reads some a = 9 and some a = 17
END TRANSACTION                                     END TRANSACTION
```

Lost Update — **Definition 7.3.7**

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                                BEGIN TRANSACTION T2
WITH old AS (SELECT a FROM table WHERE b = 1)
                                                    UPDATE table SET a = 9 WHERE  b = 1;
UPDATE table SET a = (
    SELECT a + 4 FROM OLD
) WHERE  b = 1;
END TRANSACTION                                     END TRANSACTION
```

7.4 Isolation Levels

Read Uncommitted — **Definition 7.4.1**

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 7.4.2**

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

Readers wait for writers to commit. All writes of a transaction are atomically made available at commit time.

- Each row is read/write locked, read locks acquired & dropped as needed, write locks held until commit/abort.

Repeatable Read **Definition 7.4.3**

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

Strengthen's read committed to guarantee any repeated read in a transaction returns the same result.

- Phantom reads can occur (different rows $\to$ different reads)
- Each row is read/write locked, both read & write locks held until commit/abort.

Serializable **Definition 7.4.4**

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

Readers get full isolation, execution is serializable.

- Restrictive & expensive $\to$ never the default isolation level.
- Each range of rows (e.g table) affected by a transaction is read/write locked for the entire transaction.

7.5 Concurrency Schemes

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

7.5.1 Serial Execution

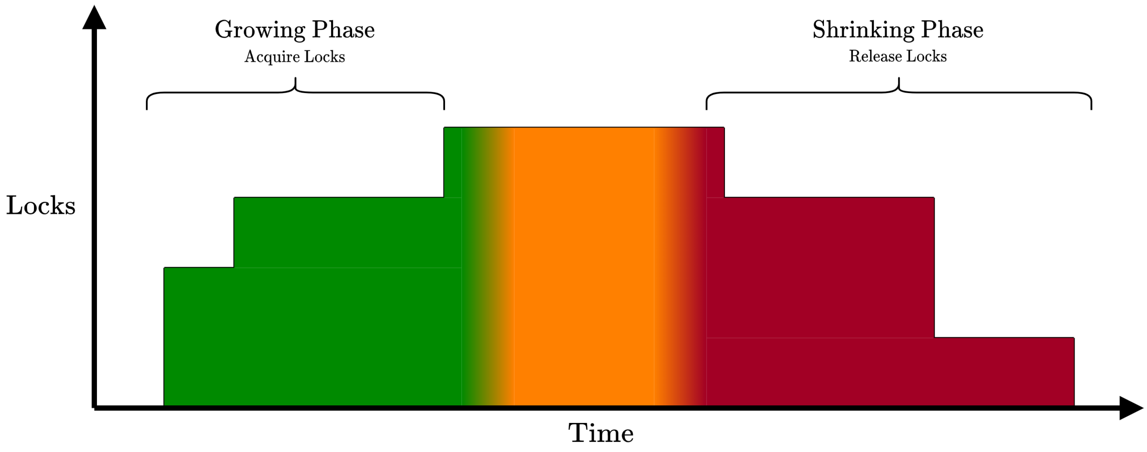
No concurrency, execute all transactions in sequential order.

No Anomalies	Solves all aforementioned anomalies.
Simple Implementation	Easy to implement: No concurrency $\Rightarrow$ No problems!

Latency	As transactions cannot occur concurrently, they must be queued and the user must wait. Very poor performance under load.
Underutilisation	Limited usage of hardware $\to$ limit to how much individual queries can be parallelised to use cores, more transactions in parallel solves this.
Not Scalable	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).

7.5.2 Two-Phase Locking (2PL)



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

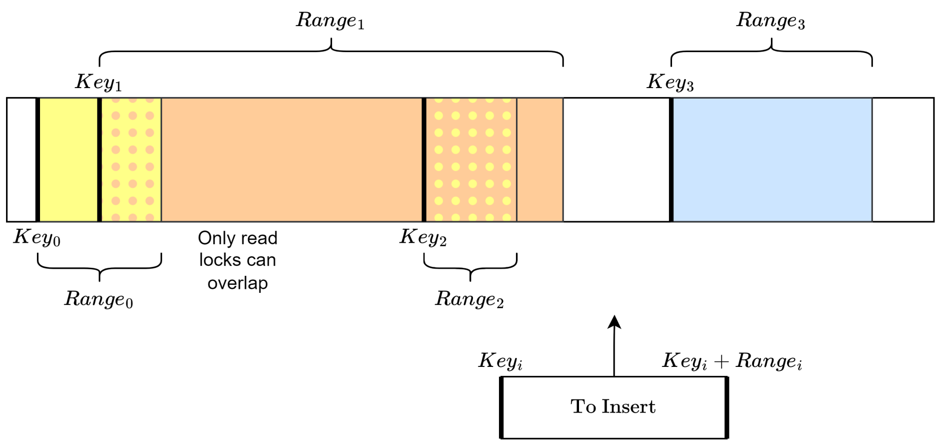
Deadlocks

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

- Acquire locks in a global order $\to$ 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 $\to$ 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

To maximise concurrency, we want to lock as little as is required for the given isolation level.

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.

Predicate Locking — ***Extra Fun!* 7.5.1**

Rather than locking objects (e.g rows, tables), lock predicates.

```sql
-- lock updates, deletions, inserts of rows potentially covered by this predicate
SELECT * FROM people WHERE name <> "bob" AND age > 18;
UPDATE people SET cool = true WHERE name <> "bob"; -- can run concurrently
```

- On attempting a query, use the current predicate locks, and predicate used for access to determine locking.
- Locking rows can prevent Non-repeatable-read, locking inserts to a table (using predicate locking) can prevent phantom read.

7.5.3 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.

7.5.4 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).

7.5.5 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 8

Streams

8.1 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

8.2 Push Operators

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

```
// For easy include of files in the notes, each operator is in a different file
// more maintainable that using line numbers with \inputminted
#include "operators/output.h"
#include "operators/project.h"
#include "operators/push_operator.h"
```

- 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).

8.2.1 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

```cpp
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.

```cpp
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.

```cpp
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 (Event 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
```

8.2.2 PushBack

Resource usage of operators is important.

- Some operators may buffer rows (in order to resolve order, retain aggates 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.

8.3 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 8.3.1**

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 | unpredicatable | low-overhead

Ingestion-Time — **Definition 8.3.2**

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 | unpredicatable | medium-overhead

Event-Time — **Definition 8.3.3**

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 | predicatable | high-overhead

8.3.1 In-Order Processing

In-Order Processing — **Definition 8.3.4**

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

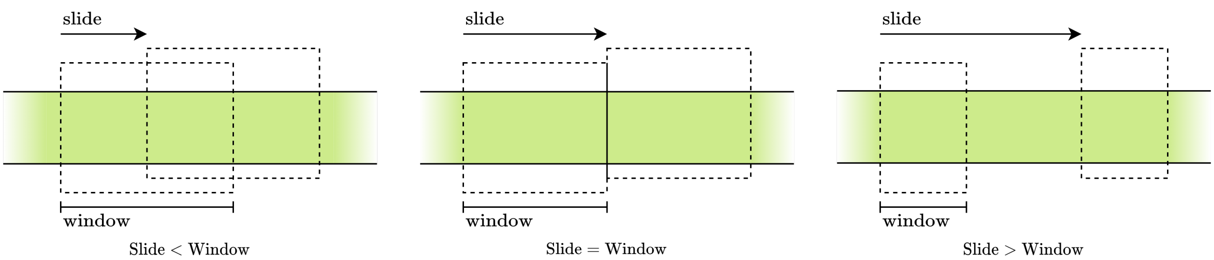
Watermarks/Punctuation

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.

8.3.2 Windows



Sliding Window Tumbling Window Stream Sampling

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!* 8.3.1**

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

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

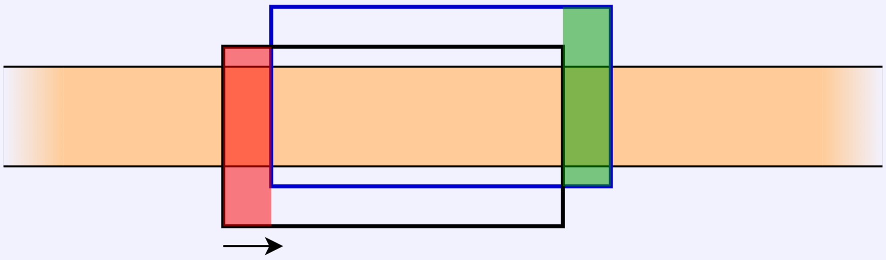
We can run aggregate functions on a window:

```
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 8.3.5**

$$sum(new\ window) = sum(window) - old + new$$



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
```

8.3.3 Aggregate Implementations

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

Window Sum

```cpp
class WindowSumAggregator : public PushOperator<float> {
  PushOperator<float> *plan_;

  // a circular buffer window
  // the next index after buffer_i_ is the start of the window
  std::vector<float> window_buffer_;

  size_t buffer_i_ = 0;
  float aggregate = 0;

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

public:
  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;
    }
  }
};
```

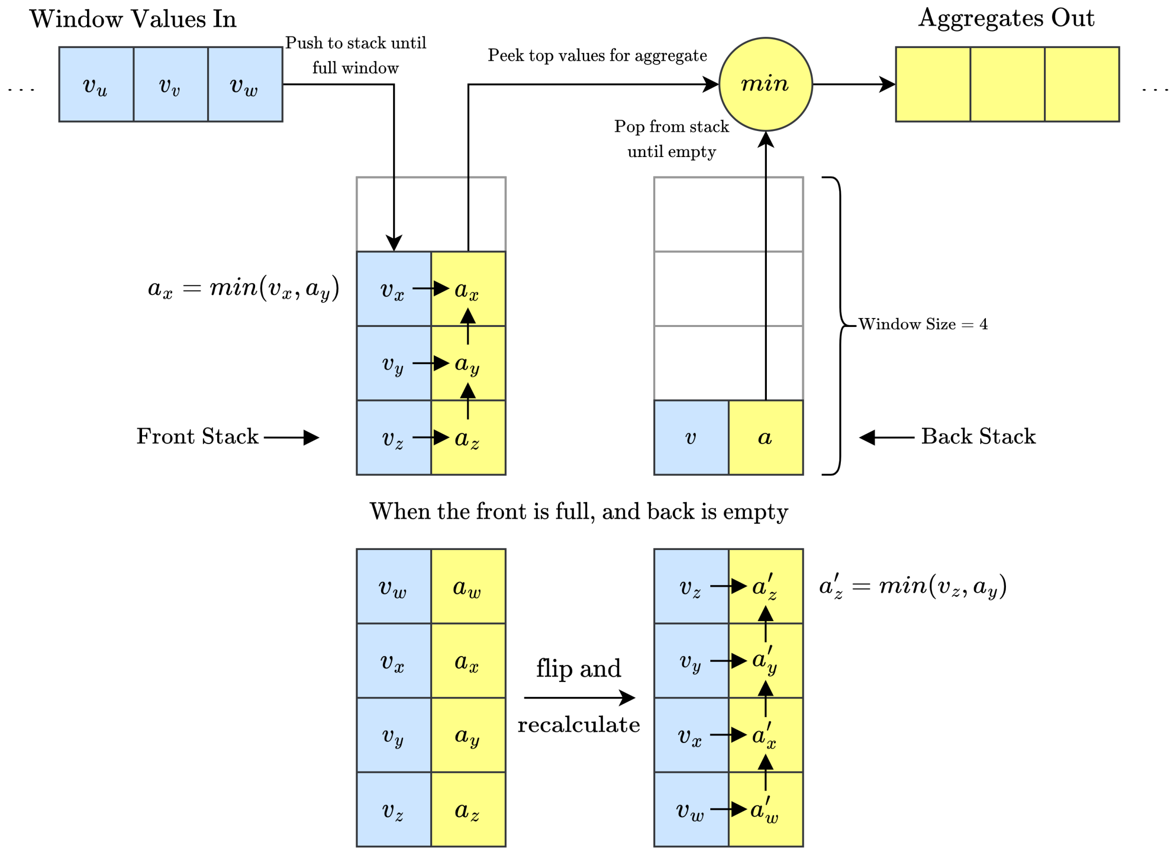
Window Median

Improve Me! ***Extra Fun!* 8.3.2**

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]);
      }
    }
  }
};
```

8.3.4 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 $\frac{1}{window\ size}$) flip the front stack, recalculate aggregates and set to back stack.

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

```
// To improve: we can use one vector instead of two separate
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
  // invariants:
  // - 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();
```

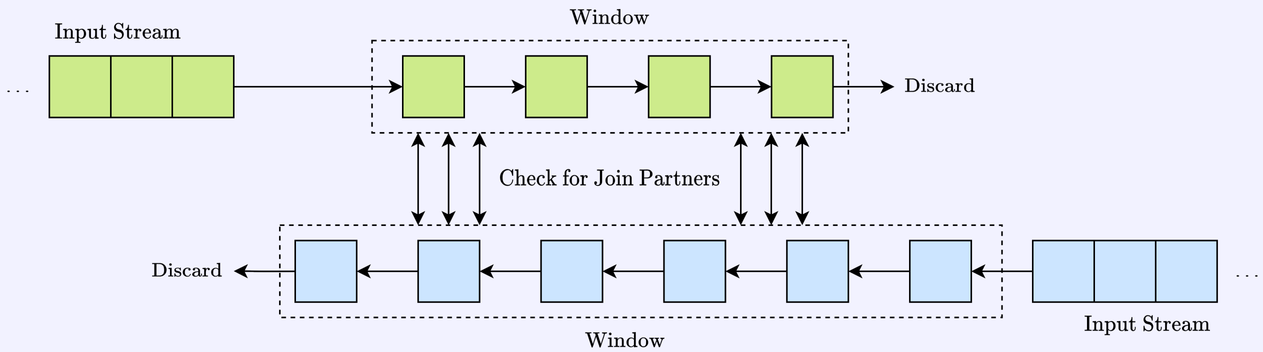
Space Efficiency — ***Extra Fun!* 8.3.3**

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).

8.4 Stream Joins

8.4.1 Handshake Join

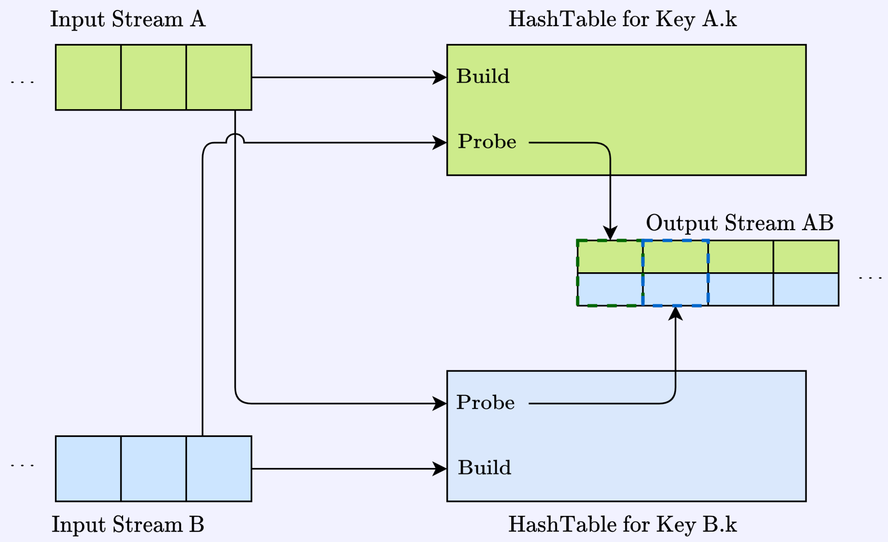
Handshake Join — **Definition 8.4.1**



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

8.4.2 Symmetric Hash-Joins

Symmetric Hash-Joins **Definition 8.4.2**

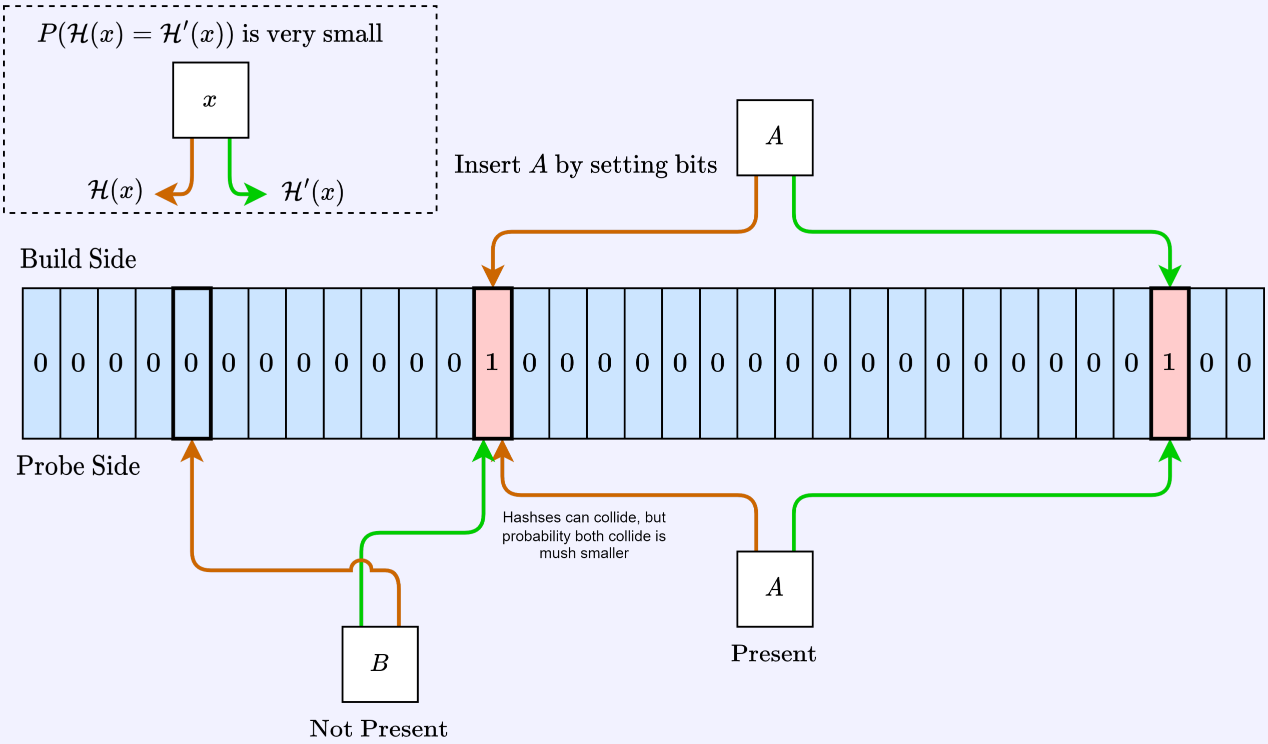


Both input streams build their own hashtable, while probing the other. Matches from probes are inserted into the joined output stream.

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

8.4.3 Bloom Filters

Bloom Filter — **Definition 8.4.3**



A table of bits, indexed by hashing the key used for the join. By using multiple independent hashes, the probability all collide is low.

- Collisions/false-positives still possible.
- Can use as many independent hashes as needed.
- Uses finite space.
- Can be used to implement a form of symmetric hash-join.

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
ϵ	False-positive rate

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

Chapter 9

Advanced Topics

9.1 Hardware and Data Models

The Turing Tax — ***Extra Fun!* 9.1.1**

The additional cost/overhead (performance, hardware cost, energy) of universality/general purpose computing in hardware.

	Example	Description
General Purpose	CPU	*Jack of all trades, but a master of none.*
Dedicated	GPU, TPU	Optimised for a very specific set of operations.

The *turing tax* & related tradeoffs of general purpose computing are discussed at length in Dr Paul Kelly's *60001 - Advanced Computer Architecture* Module.

Hardware Heterogeneity is Increasing

- The end of moore's law the *free lunch* provided decades of performance improvements by *dennard scaling* is ending.
- Dedicated accelerators for specific applications/operations can provide increased performance by avoiding/reducing the *turing tax*
- As a result, systems need to be able to efficiently use many different accelerators.

GPU — **Definition 9.1.1**

Graphics Processing Unit, designed for highly parallel operations on data (operating the same instructions across many threads in many warps).

TPU — **Definition 9.1.2**

Tensor Processing Unit, developed by Google for low precision arithmetic on tensors (matrices are 2D tensors)

ASIC — **Definition 9.1.3**

Application-specific Integrated Circuit. An IC designed to compute a specific application and hence with virtually no associated *turing tax* overhead.

Near Memory Computing — **Definition 9.1.4**

Accelerators built into/physically adjacent to main memory to avoid the bandwidth limitations of CPU memory access over a memory bus.

Field Programmable Gate Array (FPGA) — **Definition 9.1.5**

An array of programmable blocks that can be configured to a specific design (described by a developer using a *hardware description language*) to perform a specific algorithm.

Data Model Heterogeneity is Increasing

Many new data models have been developed to support specific types of application.

- Key value stores used to improve performance of distributed systems through caching.
- Graph based models for highly interconnected data (e.g social networks) that avoid the costs associated with joins on very large relations

- Document based databases for flexibility & simplicity in storing data (e.g storing BSON objects in MongoDB to support simple webapps)

Redis — *Extra Fun!* 9.1.2

Redis is a popular in-memory key value store, often used as a cache but also usable as a key-value database.

Memcached — *Extra Fun!* 9.1.3

Memcached is a distributed key-value store designed for caching. Usage is nicely explained in this funny story.

RedisGraph — *Extra Fun!* 9.1.4

A graph based database RedisGraph which uses adjacency matrices & smart linear algebra to achieve a self-proclaimed title of *fastest graph database*.

Workload Heterogeneity is Increasing

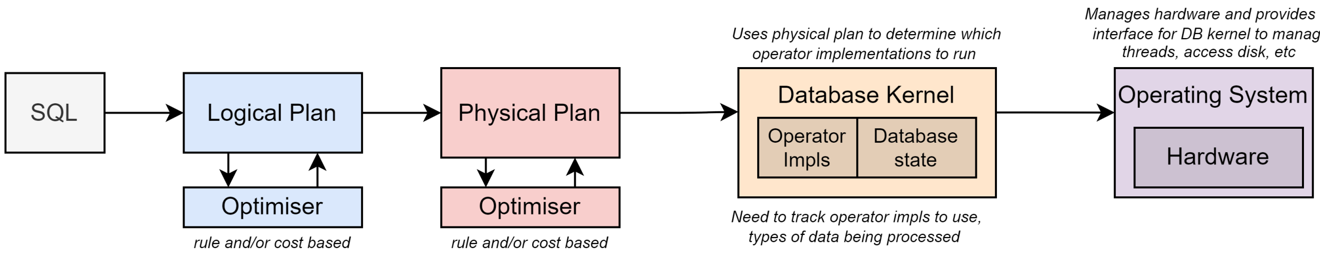
Datasets are growing larger with more kinds of workload.

Analytics   Transaction Processing   Inference   Data Cleaning   Data Integration

- Data integration workloads are required for the large distributed data systems
- Data science related workloads needed at scale (cleaning, model training and inference)

9.2 CodeGen

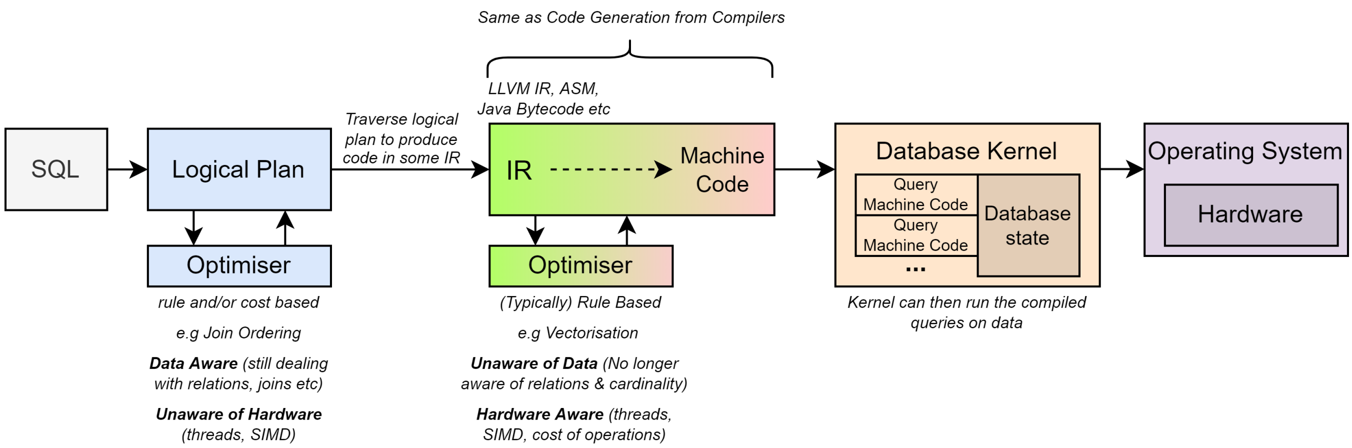
A typical DBMS implementation converts queries to logical, then physical plans. The kernel then invokes operator implementations specified in a query's physical plan to process the query.



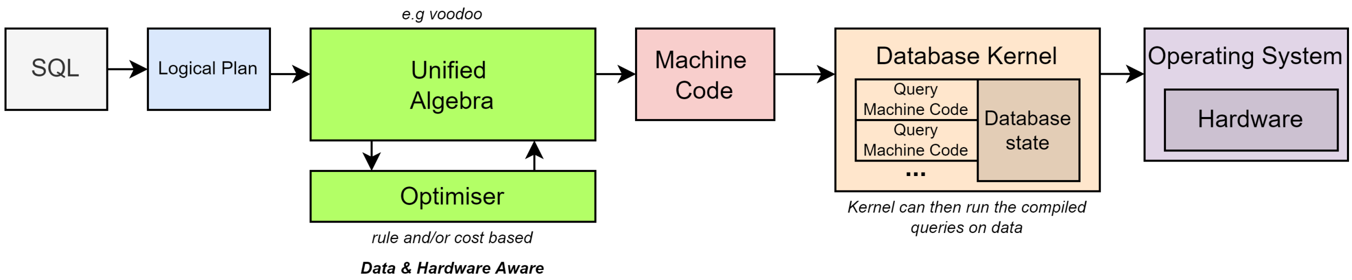
There are several unavoidable costs/limitations to optimisation:

- With volcano processing we must compose/stitch together operators at runtime, necessitating expensive virtual calls.
- Inter-Operator microarchitectural optimisations (e.g inlining volcano operators) are not possible as the kernel can only use operator implementations, not edit/optimise/restructure their code

Alternatively we could generate the code for operator implementations at query time, with all the information available at that time.



There are optimisations that require both data and hardware awareness, particularly relating to parallelism (needs to understand data dependencies as well as the parallelism supported by hardware).



voodoo — **Definition 9.2.1**

A *Vector-Dataflow Language* used as a unified algebra for code generating DBMS (original paper).

9.2.1 Vector Operations

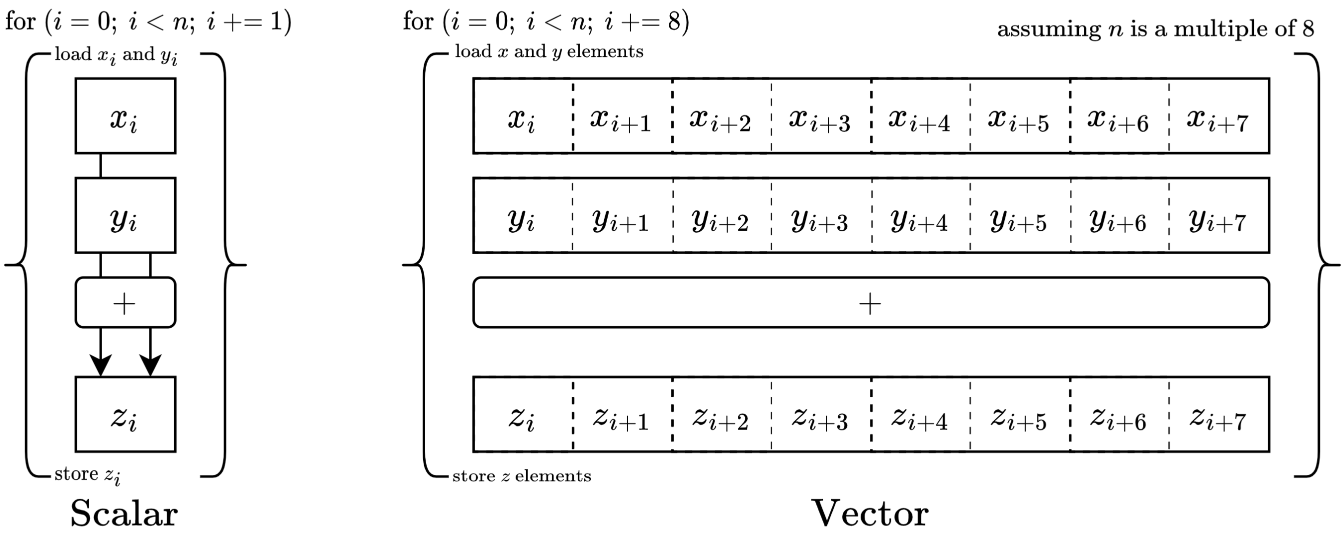
$$[x_1, \quad x_2, \quad \dots, \quad x_n] + [y_1, \quad y_2, \quad \dots, \quad y_n] = [x_1 + y_1, \quad x_2 + y_2, \quad \dots, \quad x_n + y_n]$$

A single instruction/*operation* (e.g $+$) operating on multiple data.

- As each operation is independent, each can be performed in parallel.
- A very large vector can be partitioned and processed on multiple threads.
- To take advantage of this parallelism in a single thread, we can use vector extensions.
- Vector extensions include wider registers, and special instructions for operating on *lanes* of a vector register in parallel.

For example element-wise sum over two tables (e.g previously joined in the plan).

```
CREATE TABLE numbers (x BIGINT, y BIGINT); /* ... */; SELECT (x + y) as z FROM numbers;
```



Naively we could generate some *scalar* code to perform the operation.

```
template<size_t n>
void element_sum_scalar(int64_t x[n], int64_t y[n], int64_t z[n]) {
    for (auto i = 0; i < n; i++) z[i] = x[i] + y[i];
}
// Note: with optimisation on clang & gcc will automatically vectorize this
```

We could use multithreading.

```cpp
template <size_t n>
void element_sum_threads(int64_t x[n], int64_t y[n], int64_t z[n]) {
    //number of concurrent threads supported
    const auto no_threads = std::thread::hardware_concurrency();

    // round-up integer division to get elements computed per thread
    const auto n_per_thread = ((n - 1) / no_threads) + 1;

    std::vector<std::thread> threads;

    for (auto index = 0; index < n; index += n_per_thread) {
        threads.emplace_back([&, index] {
        for (auto s = index; s < std::min(index + n_per_thread, n); s++)
            z[s] = x[s] + y[s];
        });
    }

    for (auto &t : threads) t.join();
}
```

Or we can use a vector extension we know is available on the hardware the system is running on, such as AVX-512 used here.

```cpp
#include <immintrin.h> // intel intrinsics used to ensure we use 512 bit vector instructions
#include <type_traits> // using enable_if as this code only works for n that are multiples of 8

template<size_t n>
typename std::enable_if<n % 8 == 0, void>::type
element_sum_vec(int64_t x[n], int64_t y[n], int64_t z[n]) {
    for (auto i = 0; i < n; i+=8) {
        __m512i xs = _mm512_loadu_si512(&x[i]);
        __m512i ys = _mm512_loadu_si512(&y[i]);
        __m512i zs = _mm512_add_epi64(xs, ys);
        _mm512_storeu_si512(&z[i], zs);
    }
}
```

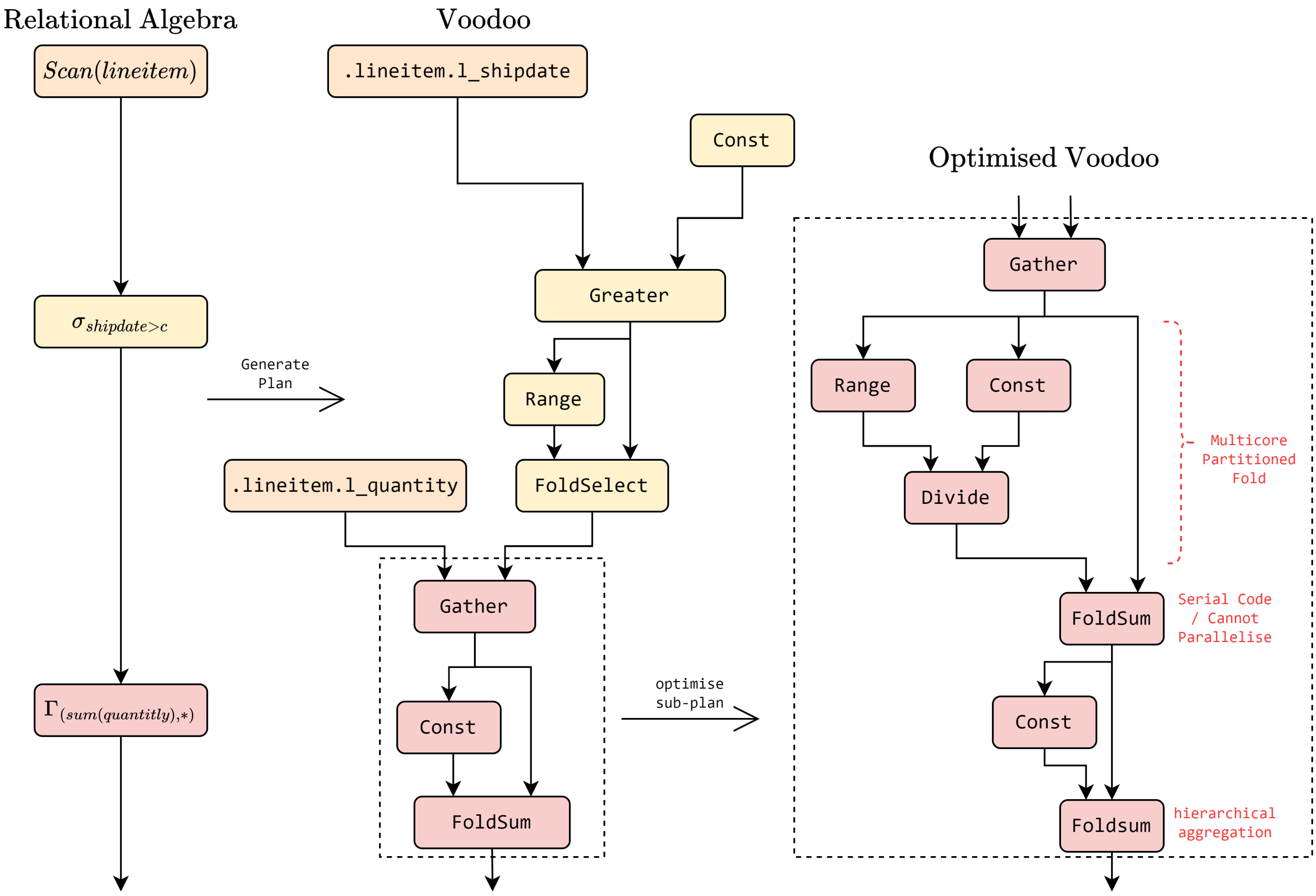
Given some call to `element_sum_vec<2048>(x, y, z)` we can compile:

```
g++ -O3 -mavx512f vectorisation.cpp  # Compiled with mavx512f to let GCC use AVX-512 instructions
```

And extract the loop doing the summation:

```asm
# Arrays stack allocated
.element_sum_vec:
    xor     eax, eax
.Loop:
    vmovdqu64  zmm1, ZMMWORD PTR [rsp+rax]          # xs = x[i:i+8]
    vpaddq     zmm0, zmm1, ZMMWORD PTR [r13+0+rax]  # zs = xs + y[i:i+8]
    vmovdqu64  ZMMWORD PTR [rbx+rax], zmm0          # z[i:i+8] = zs

    add rax,    64    # i += 8    * sizeof(int64_t)
    cmp rax, 16384    # i != 2048 * sizeof(int64_t)
    jne .Loop
    ret
```

9.2.2 Data Flow



Voodoo expresses plans as a data-flow graph containing vector operations (which it can parallelise with multithreading, SIMD or GPU).

9.3 Adaptive Indexing

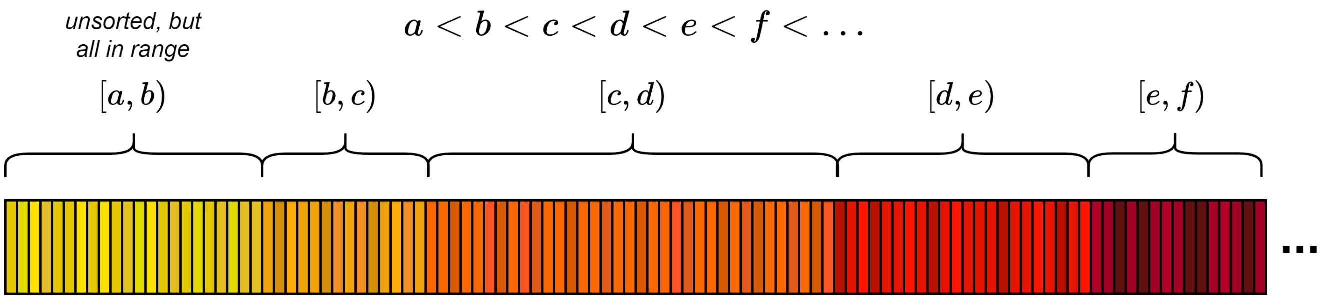
9.3.1 Cracking

```
SELECT * FROM table WHERE x BETWEEN c1 AND c2; -- Given constants c1 and c2
```

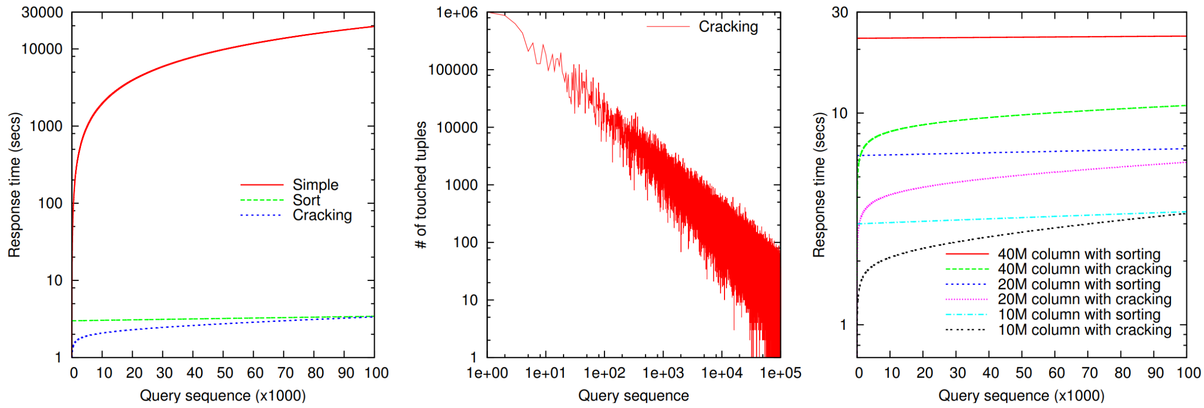
Scan Linearly scan table, get entries.

Sorted Index Build a sorted index, maintain the index under writes, inserts & deletes. Use index to get range efficiently.

Cracking Split/*crack* the table into several unsorted ranges, with the ranges in sorted order.



Cracking has been shown to significantly improve performance, as examined in the paper **Database Cracking** (benchmarking cracking in monetDB).

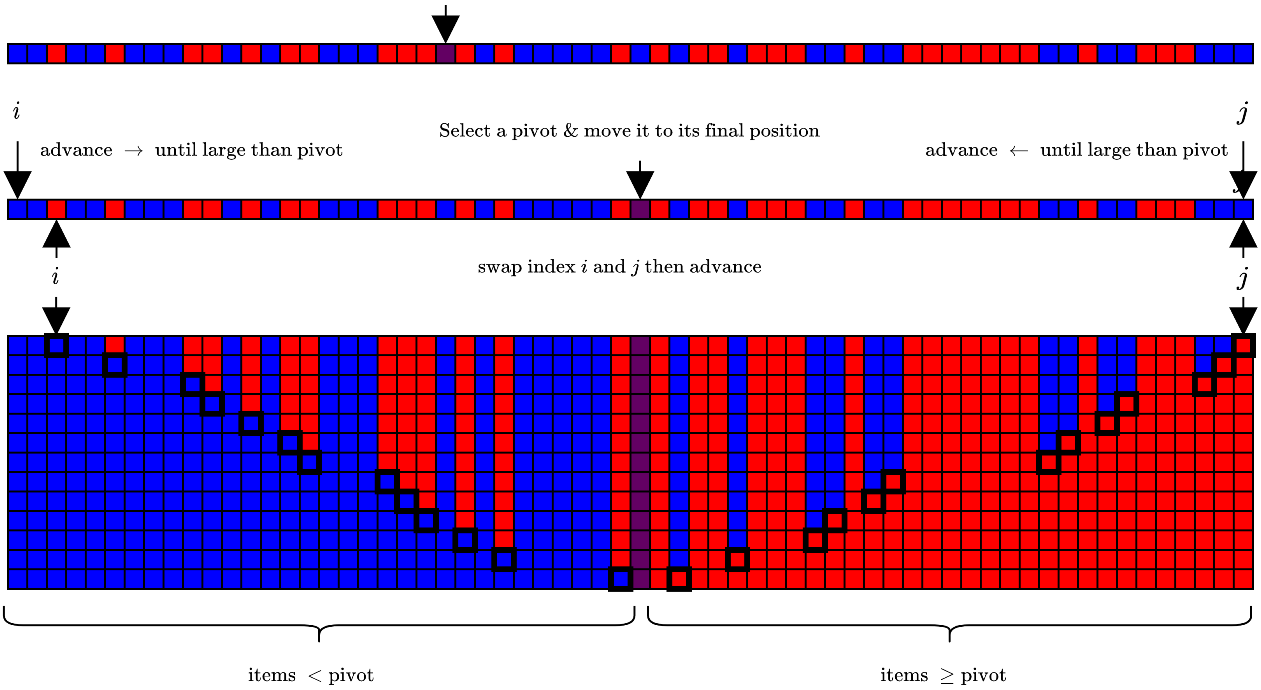


(a) Selecting on a 10 million column (b) Portion touched by crack select (c) Cracking larger columns

"Cracking stuff Gromit!"	*Extra Fun!* 9.3.1

- These slides cover the cracking implementation in monetDB.
- Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores

9.3.2 Hoare Partitioning



The partitioning algorithm used in quicksort

- $O(n)$ time complexity
- Does not require extra memory / partitions in-place.

```
namespace hoare {
// INV: sort_vec.size() > 0
template <typename T>
size_t partition(std::vector<T> &sort_vec, size_t start, size_t end) {
  // get pivot
  T pivot = sort_vec[start];
  size_t count = 0;
```

```
  // determine where to partition / where to place pivot value
  for (size_t i = start + 1; i < end; i++) {
    if (sort_vec[i] <= pivot)
      count++;
  }

  // swap pivot into place, will partition around pivot
  size_t pivotIndex = start + count;
  std::swap(sort_vec[pivotIndex], sort_vec[start]);

  // start pointers i & j at ends of range
  size_t i = start, j = end - 1;

  // advance pointers, swap and partition
  while (i < pivotIndex && j > pivotIndex) {
    while (sort_vec[i] <= pivot)
      i++;
    while (sort_vec[j] > pivot)
      j--;

    if (i < pivotIndex && j >= pivotIndex) {
      std::swap(sort_vec[i], sort_vec[j]);
      i++;
      j--;
    }
  }

  return pivotIndex;
}
} // namespace hoare
```

Consider the following section from the algorithm.

```
while(sort_vec[i] <= pivot) i++;
```

A *hot loop* containing a conditional with low selectivity (on random data).

- we can employ an out-of-place algorithm that allows us to remove this *control hazard*

9.3.3 Predication

Predication — ***Extra Fun!* 9.3.2**

The idea behind predication is to avoid *control hazards*, rather than branching to different instructions, just conditionally apply instructions. Some architectures support this directly with predicated instructions, such as IA64 (full predication) or ARM.

```
if (a > b) a += b;
```

```
    cmp r0, r1                    cmp r0, r1
    addlt r0, r0, r1              ble dont_add
                                  add r0, r0, r1
                              dont_add:
                                  @ ...
```

In the below examples we *predicate* by removing a jump/branch, rather than using predicated instructions.

We can start with a basic out-of-place partition.

```
namespace out_of_place_cond {
// INV: input_vec.size() > 0
template <std::copy_constructible T>
size_t partition(const std::vector<T> &input_vec, std::vector<T> &output_vec,
                 size_t start, size_t end) {
```

```
  const T &pivot = input_vec[(start + end) / 2];
  size_t left_index = start;
  size_t right_index = end - 1;

  for (auto i = start; i < end; i++) {
    if (input_vec[i] < pivot) {
      output_vec[left_index] = input_vec[i];
      left_index++;
    } else {
      output_vec[right_index] = input_vec[i];
      right_index--;
    }
  }

  return left_index;
}
} // namespace out_of_place_cond
```

Here the `if (input_vec[i] < pivot)` condition has low selectivity, and is part of the hot loop.

We can predicate this by always writing the `input_vec[i]`, and incrementing the pivot indexes based on the condition.

```
namespace out_of_place_pred {
// INV: input_vec.size() > 0
template <std::copy_constructible T>
size_t partition(const std::vector<T> &input_vec, std::vector<T> &output_vec,
                 size_t start, size_t end) {
  const T &pivot = input_vec[(start + end) / 2];
  size_t left_index = start;
  size_t right_index = end - 1;

  for (auto i = start; i < end; i++) {
    output_vec[left_index] = input_vec[i];
    output_vec[right_index] = input_vec[i];

    // increment using boolean, if not incremented, value is overwritten on
    // the next iteration of the loop
    left_index += input_vec[i] < pivot;
    right_index -= input_vec[i] >= pivot;
  }

  return left_index;
}
} // namespace out_of_place_pred
```

9.3.4 Predicated Cracking

Cracking stuff! ***Extra Fun!* 9.3.3**

The algorithm presented here is from the paper *Database Cracking: Fancy Scan, not Poor Man's Sort!*

```
namespace predicated_cracking {

constexpr bool USE_CONDITIONS = false;

// INV: sort_vec.size() > 0
template <typename T>
size_t partition(std::vector<T> &sort_vec, size_t start, size_t end) {

  bool cmp = false;
```

```
  size_t left_ptr = start;
  size_t right_ptr = end - 1;
  T active = sort_vec[left_ptr];
  T backup = sort_vec[right_ptr];
  T pivot = sort_vec[(start + end) / 2]; // somewhat arbitrary pivot selection

  while (left_ptr < right_ptr) {

    // write active
    sort_vec[left_ptr] = active;
    sort_vec[right_ptr] = active;

    if constexpr (USE_CONDITIONS) {
      if (pivot > active) {
        left_ptr++;
        active = sort_vec[left_ptr];
      } else {
        right_ptr--;
        active = sort_vec[right_ptr];
      }
    } else {
      // compare and write
      cmp = pivot > active;

      // advance cursor
      left_ptr += cmp;
      right_ptr -= 1 - cmp;

      // backup phase
      active = cmp * sort_vec[left_ptr] + (1 - cmp) * sort_vec[right_ptr];
    }

    // swap active
    std::swap(active, backup);
  }

  sort_vec[left_ptr] = active;
  return left_ptr;
}

} // namespace predicated_cracking
```

Conditional Understanding — ***Extra Fun!* 9.3.4**

A python script in the included code for these notes can be used to generate apply and print steps of this algorithm.

Chapter 10

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.