60029

Data Processing Systems

Imperial College London

Contents

Chapter 1

Introduction

1.1 Logistics

A note on types... *Extra Fun!* **1.1.1**

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

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

1.2 Data Management Systems

Database **Definition 1.2.1**

A large collection of organized data.

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

System **Definition 1.2.2**

A collection of components interacting to achieve a greater goal.

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

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

Data Management System **Definition 1.2.3**

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

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

Data Processing System **Definition 1.2.4**

A system for processing data.

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

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

Building data management systems is hard!

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

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)
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.
User Defined Functions	
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.

There are several common patterns for data-intensive applications:

Online Transaction Processing (OTP)

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

Goal: Throughput

Online Analytical Processing (OLAP)

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

Goal: Latency

Reporting

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

Goal: Resource Efficiency

Daily Struggle **Example Question 1.3.1**

Provide some examples of *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.

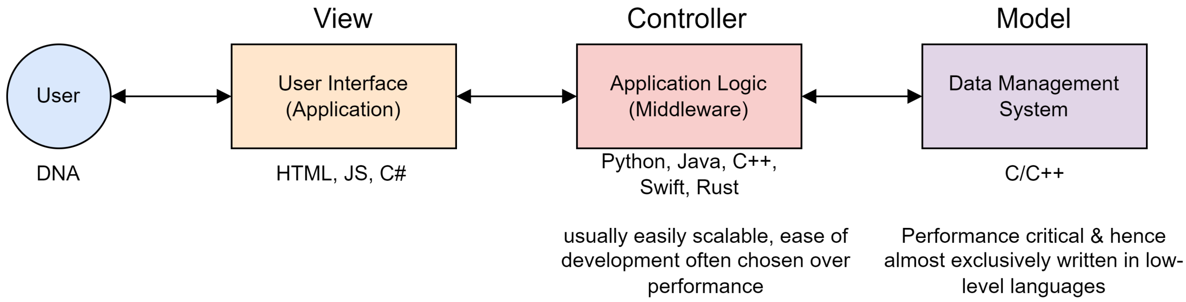
Data-Intensive Applications can be differentiated from *Data Management Systems* (though there is ample ambiguity):

- 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 1.3.2**

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

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



Big Business *Extra Fun!* **1.3.2**

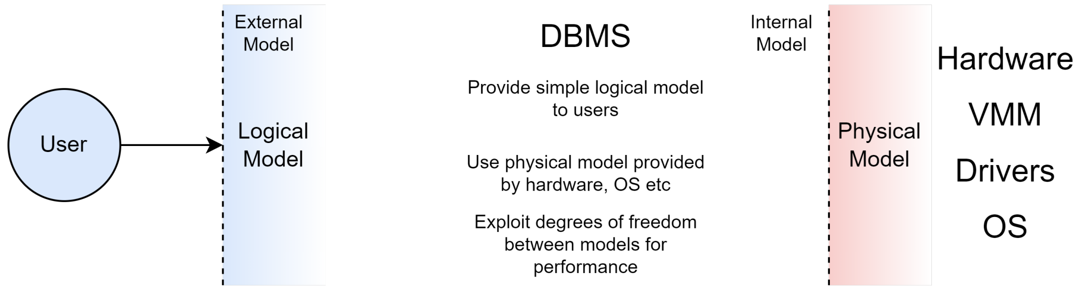
The enterprise data management systems market has been valued at \$82.25 billion (2021) with annual growth exceeding 10% (grand view research).

1.4 Data Management Systems

1.4.1 Non-Functional Requirements

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



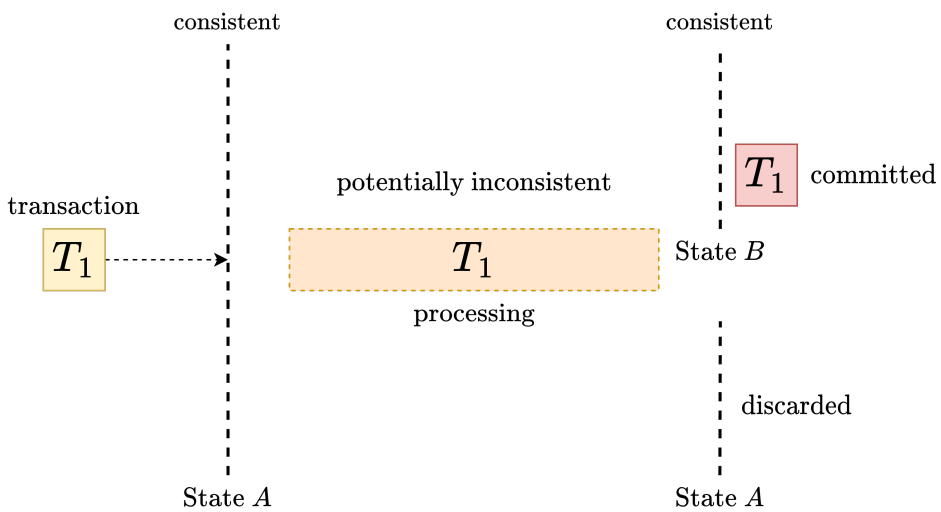
1.4.3 Transactional Concurrency

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.

1.4.4 Read Phenomena

Dirty Read / Uncommitted Dependency — **Definition 1.4.2**

A transaction reads a record updated by a transaction that has not yet committed.

- The uncommitted transaction may fail or be rolled back rendering the dirty-read data invalid.

Non-Repeatable Read — **Definition 1.4.3**

When a transaction reads a record twice with different results (another committed transaction updated the row between the reads).

Phantom Reads — **Definition 1.4.4**

When a transaction reads a set of records twice, but the sets of records are not equal as another transaction committed between the reads.

1.4.5 Isolation levels

Serialisable — **Definition 1.4.5**

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)

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.

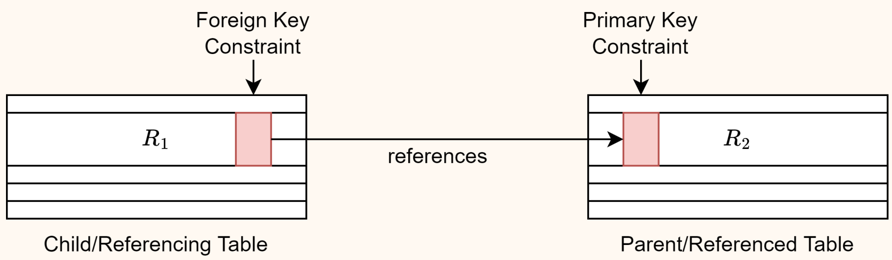
  ```
  CREATE TABLE foo (bing INTEGER, zog TEXT, bar INTEGER);
  ```

- 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).

$$
\begin{array}{l|cccc|l}
\text{Columns:} & \text{X-ray} & \text{Yankee} & \dots & \text{Zulu} & \\
 & (x_1 & y_1 & \dots & z_1) & \\
 & (x_2 & y_2 & \dots & z_2) & \\
 & (x_3 & y_3 & \dots & z_3) & \text{Unique} \\
 & (x_4 & y_4 & \dots & z_4) & \\
 & \vdots & \vdots & \vdots & \vdots & \\
 & (x_n & y_n & \dots & z_n) & \\
 & \multicolumn{4}{c}{\text{Type } (X, Y, \dots, Z)} &
\end{array}
$$

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

Project Select Cross/Cartesian product Union Difference

Relational algebra is closed:

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

Query This! — **Example Question 2.1.2**

Given the below structure, write a query. **UNFINISHED!!!**

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

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

Here a natural join is:

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

Unique Addresses — **Example Question 2.1.3**

UNFINISHED!!!

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

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

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

2.1.2 Nomenclatures

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

2.2 Implementing Relational Algebra in C++

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

```cpp
#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.

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

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

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

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

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

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

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

2.2.1 Relation

```cpp
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 to

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

  // a variant is a type safe union. It is either a function on rows, or a
  // mapping of columns
  variant<function<tuple<outputTypes...>(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!* 2.2.1**

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

2.2.3 Select

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

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

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

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

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

```cpp
struct Condition {
  Comparator compare;

  Column leftHandSide;
  variant<Column, Value> rightHandSide;

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

Enums vs Enum classes *Extra Fun!* **2.2.2**

`enum class`	`enum`
Enumerations are in the scope of the class No implicit conversions.	Enumerations are in the same scope as the enum Implicit conversions to integers.

Enum classes are generally preferred over enums due to the above differences.

2.2.4 Cross Product / Cartesian

$$R_1 \times R_2$$

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

```cpp
// 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.

```cpp
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){};
};
```

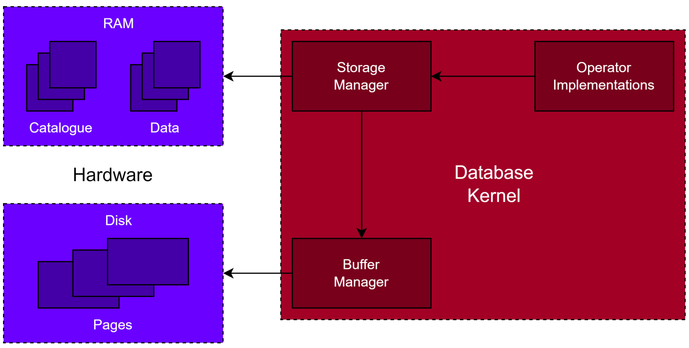
Chapter 3

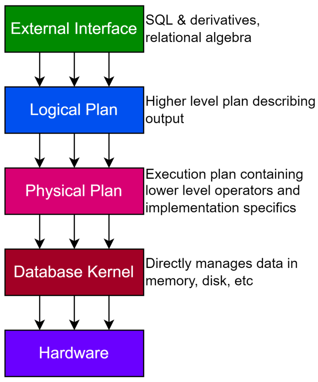
Storage

Great Exceptions! — ***Extra Fun!* 3.0.1**

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

3.1 Database Management System Kernel





Database Kernel — **Definition 3.1.1**

The core of the database management system.

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

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

3.2 Storage

3.2.1 Storage Manager

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

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

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.

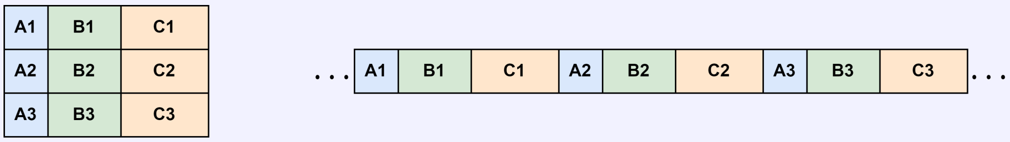
Locality — Definition 3.2.1

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

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 3.2.2

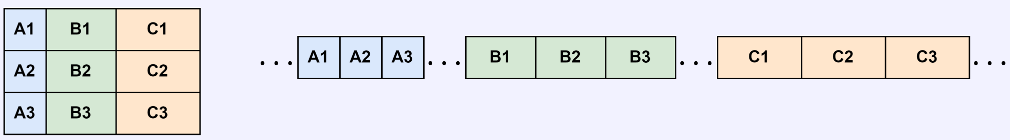
Tuples are stored adjacently.



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

Decomposed Storage — Definition 3.2.3

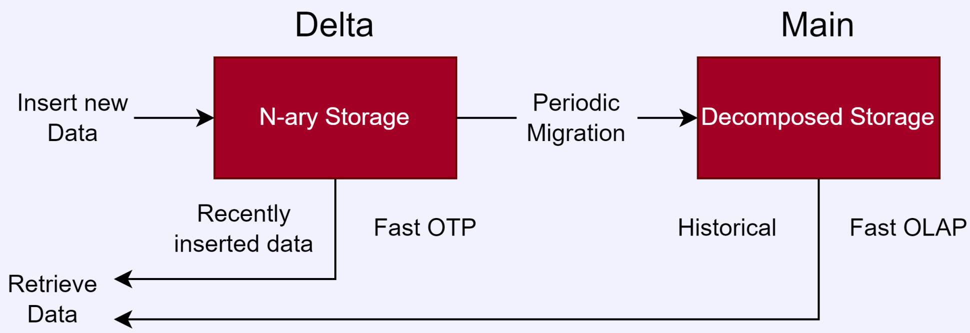
Each field of the tuple is stored in a separate array.



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

Delta/Main — **Definition 3.2.4**

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



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

3.2.2 Catalog

Catalog — **Definition 3.2.5**

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

Dense — **Definition 3.2.6**

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

3.2.3 Disk Storage

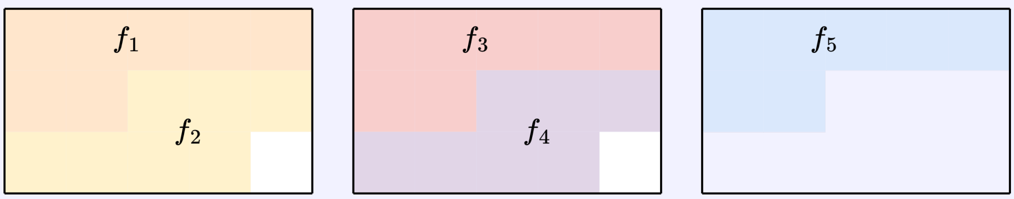
Buffer Manager — **Definition 3.2.7**

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

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

Unspanned Pages — **Definition 3.2.8**

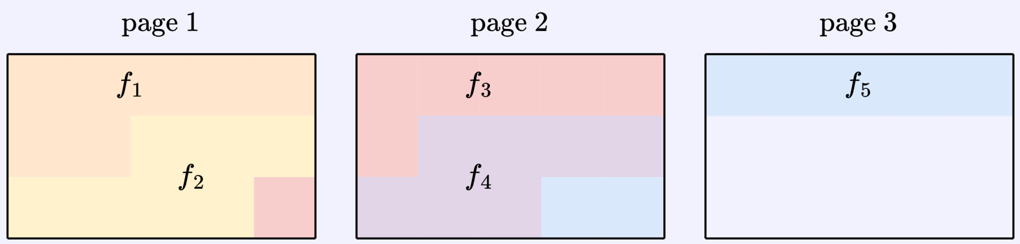
Records only allocated on the page is there is space.



- Space wasted (larger as tuple size increases)
- If the record size $>$ page size, it is not possible to use this strategy
- If records are variable size, no constant time random access.

Spanned Pages — Definition 3.2.9

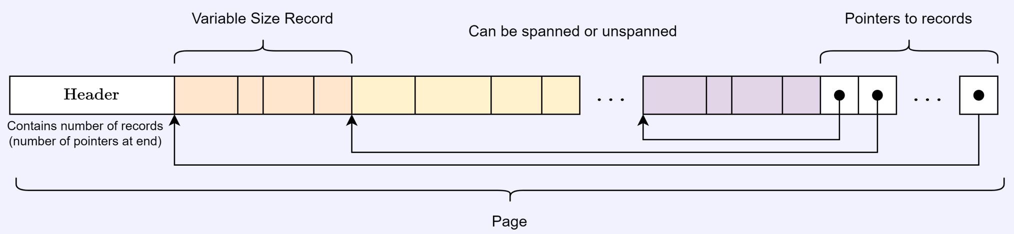
Records placed across page boundaries.



- Minimises wasted space
- Supports very large record sizes (larger than a page)
- Complex to implement, and reduced random access performance
- No in-page random access for variable size records

Slotted Pages — Definition 3.2.10

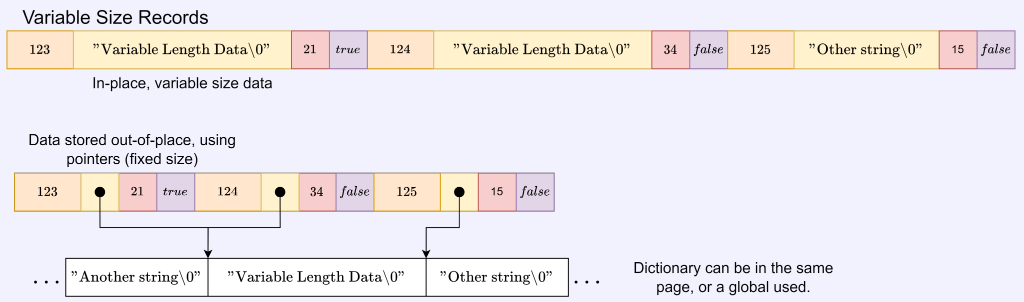
To allow faster/constant time lookup for variable size records.



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

Dictionaries — Definition 3.2.11

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).

Lookups **Example Question 3.2.1**

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

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

UNFINISHED!!!

3.3 Example Sketch Implementation

UNFINISHED!!!

Chapter 4

Algorithms and Indices

4.1 Sorting Algorithms

4.1.1 Quicksort

4.1.2 Merge Sort

4.1.3 Tim Sort

4.1.4 Radix Sort

4.1.5 Top-N with Heaps

UNFINISHED!!!

4.2 Joins

4.2.1 Database Normalisation

UNFINISHED!!!

4.2.2 Join Types

Normalised databases naturally require joins to re-compose data.

We would be honoured if you would join us. . . — **Example Question 4.2.1**

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

UNFINISHED!!!

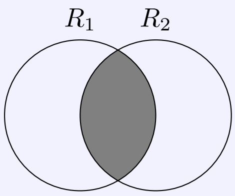
Join — **Definition 4.2.1**

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

Inner Joins

Inner Join — **Definition 4.2.2**

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



Natural Join — **Definition 4.2.3**

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

$$R_1 \bowtie R_2$$

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

Theta Join — **Definition 4.2.4**

Joining two tables based on a condition/predicate θ.

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

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

Equi Join — **Definition 4.2.5**

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

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

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

Cross Join — **Definition 4.2.6**

Just cartesian product with no selection.

$$R_1 \times R_2$$

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

Anti Join — **Definition 4.2.7**

A **theta join** using an inequality predicate

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

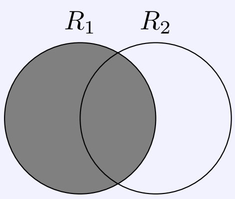
```sql
FROM R1 JOIN R2 ON R1.x <> R2.x
```

Outer Joins

Left Join — **Definition 4.2.8**

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

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

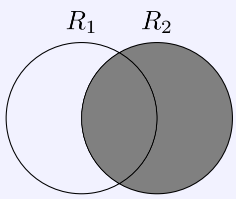



```sql
FROM R1 LEFT JOIN R2 ON ...
```

Right Join — **Definition 4.2.9**

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

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

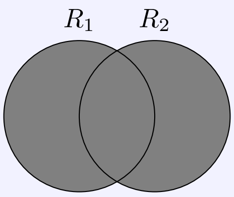



```sql
FROM R1 RIGHT JOIN R2 ON ...
```

Full Outer Join **Definition 4.2.10**

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

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




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

Which imposter? **Example Question 4.2.2**

Which of the following are joins?

1. ```
   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)

4.2.3 Join Implementations

The following join implementations are written is C++20

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

using namespace std;

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

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

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

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

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

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

4.2.4 Nested Loop Join

We can implement a basic join naively using nested loops.

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

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

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.

4.2.5 Sort Merge Join

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

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

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

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

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

```
  auto leftIndex = 0;
  auto rightIndex = 0;

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

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

  return result;
}
```

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

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.

4.2.6 Hash Join

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

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

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

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

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

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

```
  }

  return result;
}
```

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

- 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 (assumning 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).

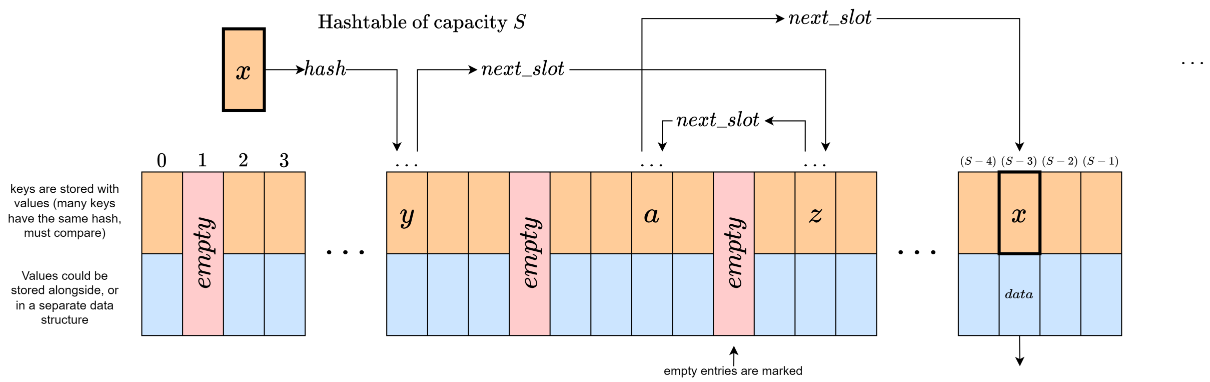
Bucket Based Hashmaps ***Extra Fun!* 4.2.1**

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

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

4.3 Hash Tables

4.3.1 Probing Hashmap



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

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

Requirement	**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)

Hash it out **Example Question 4.3.1**

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

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

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

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

Linear Probing

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

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

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

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

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

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

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

Quadratic Probing

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

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

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

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

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

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

Simple Locality	Easy to reason about memory access pattern. 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> struct ReHashProbe : public Probe<K> {
  ReHashProbe(std::function<size_t(K)> hash,
              std::function<size_t(size_t)> rehash)
      : _hash(hash), _rehash(rehash) {}

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

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

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

Simple	To implement
Reuse	Can potentially reuse the hashing function.

Locality	is poor as probes distributed uniformly.
Conflict Probability	is constant (every probe may conflict with another element).

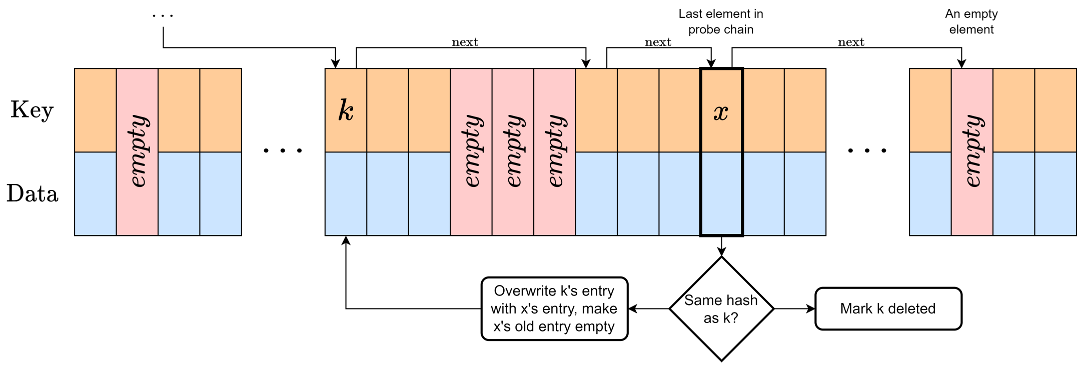
Resizing

For the example above we have considered fixed-size 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



An implementation is included below:

4.3.2 Basic Hash Table Implementation

Contribute! ***Extra Fun!* 4.3.2**

This basic implementation can be improved!

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

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

using namespace std;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

          auto endpos = lastpos(position);

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```cpp
  size_t _firstPosition;
  size_t _step;
  size_t _indexSize;
};

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

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

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

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

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

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

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

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

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

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

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

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

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

4.3.3 Partitioning

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

$$c = \text{cost of page-in}$$
$$\frac{n}{pagesize_{OS}} \times c = \text{cost of sequentially accessing } n \text{ elements}$$
$$\frac{c}{pagesize_{OS}} = \text{cost of one access}$$

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

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

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

UNFINISHED!!!

4.3.4 Indexing

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

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

Clustered/Primary Index **Definition 4.3.1**

An index storing all tuples of a table.

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

Unclustered/Secondary Index **Definition 4.3.2**

Used to store pointers to tuples of a table.

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

SQL Indexes

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

```sql
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.

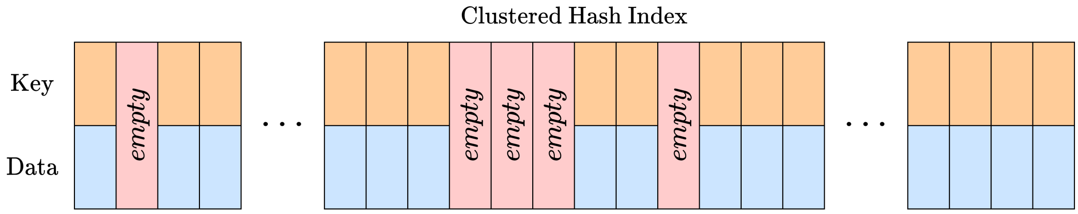
The elephant in the room *Extra Fun!* **4.3.3**

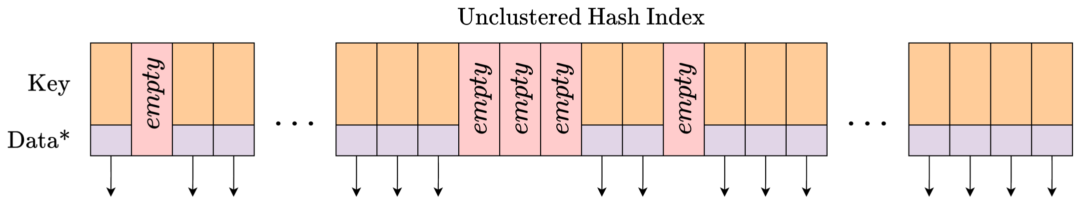
Among other DBMS, Postgres supports many types of index (documentation here)

```
/* 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');
```

4.3.5 Hash Indexes

An index backed by a hash table.





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

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

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.

4.3.6 Bitmap Indexing

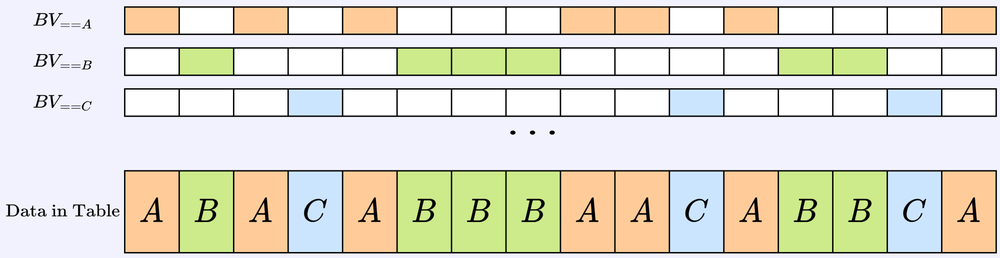
Bit Vector **Definition 4.3.3**

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

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

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

Bitmap Index — **Definition 4.3.4**



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

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

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

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.

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

using namespace std;

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

Bandwidth	Can scan a column with reduce memory bandwidth (e.g integers $\rightarrow$ 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 → 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 4.3.5**

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

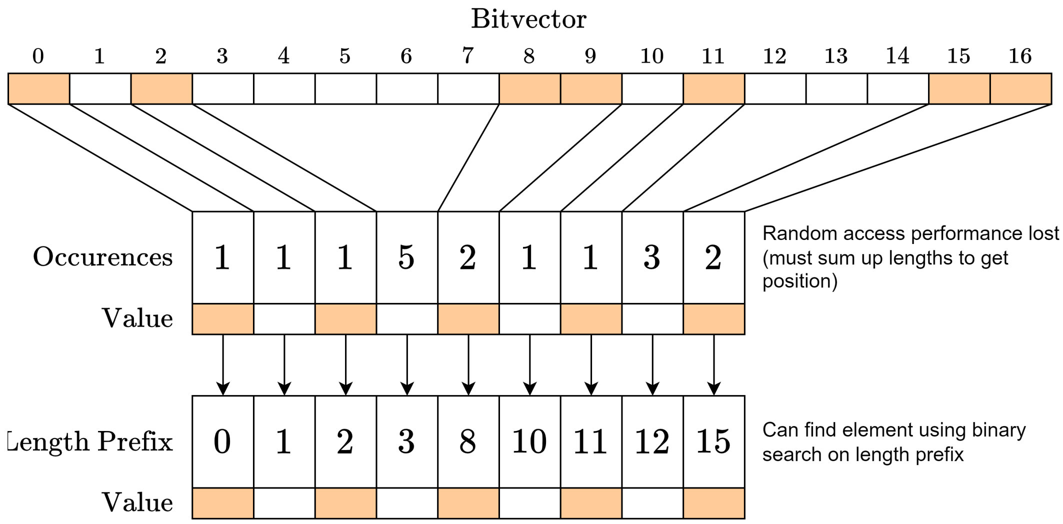
Split range into several equal size bins. Useful for uniformly distributed (or near) data.

Height Binning **Definition 4.3.6**

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



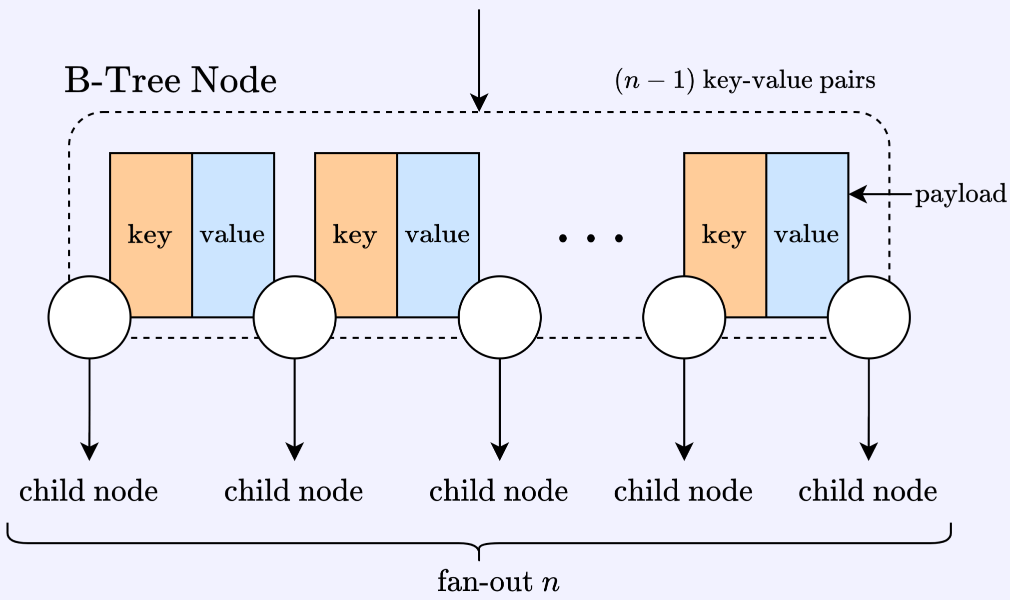
4.3.7 B-Trees

Trees are well suited to the requirements of a database:

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

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

B-Tree **Definition 4.3.7**



A high fan-out, balanced tree.

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

4.3.8 B* Trees

Maintaining Balance

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

4.3.9 Foreign Key Indices

Most joins are using a foreign key relation.

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

Chapter 5

Velox

5.1 Motivation

5.2 Overview

5.2.1 Structure

5.3 Use Cases

5.4 Library Components

5.4.1 Data Types

UNFINISHED!!!

Chapter 6

Processing Models

6.1 Motivation

Processing Model **Definition 6.1.1**

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

- Choice is critical to database design.

Function Objects **Definition 6.1.2**

References to code that can be passed, invoked, change state and produce values.

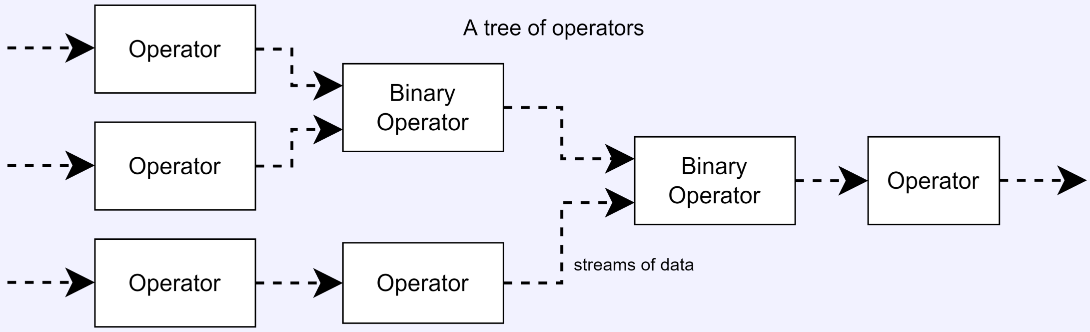
```cpp
#include <functional>

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

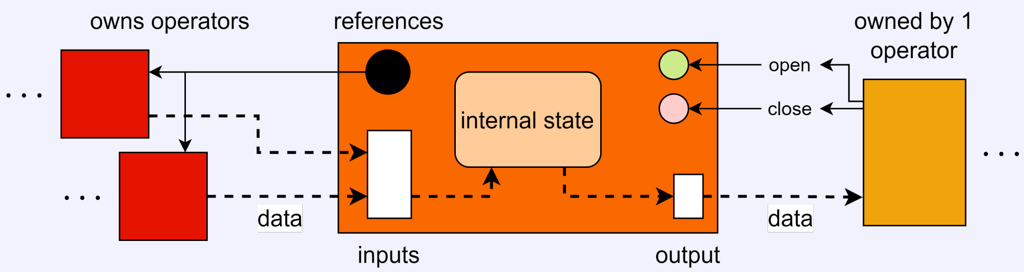
See C++11 Lambdas

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

6.2 Volcano Processing

Volcano Processing Model **Definition 6.2.1**





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).

6.2.1 Operators

A basic interface for operators can be devised as:

```cpp
#include <optional>
#include <tuple>
#include <variant>
#include <tuple>
#include <string>

using namespace std;

/* Variant used -> types for columns only known at runtime */
using Row = tuple<variant<int, double, string>>;

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

But why not RAII ***Extra Fun!* 6.2.1**

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.

Pipeline Breaker **Definition 6.2.2**

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

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