

60029

Data Processing Systems
Imperial College London

Contents

| | | |
|----------|--|----------|
| 1 | Introduction | 2 |
| 1.1 | Data Management Systems | 2 |
| 1.2 | Data Intensive Applications | 3 |
| 1.3 | Data Management Systems | 4 |
| 1.3.1 | Non-Functional Requirements | 4 |
| 1.3.2 | Logical/Physical Data Model Separation | 4 |
| 1.3.3 | Transactional Concurrency | 5 |
| 1.3.4 | Read Phenomena | 5 |
| 1.3.5 | Isolation levels | 6 |
| 1.3.6 | Declarative Data Analysis | 6 |
| 2 | Relational Algebra | 7 |
| 2.1 | Relational Structures | 7 |
| 2.1.1 | Preliminaries | 7 |
| 2.1.2 | Nomenclatures | 9 |
| 2.2 | Implementing Relational Algebra in C++ | 9 |
| 2.2.1 | Relation | 9 |
| 2.2.2 | Project | 10 |
| 2.2.3 | Select | 10 |
| 2.2.4 | Cross Product / Cartesian | 11 |
| 2.2.5 | Union | 11 |
| 2.2.6 | Difference | 11 |
| 2.2.7 | Group Aggregation | 12 |
| 2.2.8 | Top-N | 12 |

Chapter 1

Introduction

1.1 Data Management Systems

| Database | Definition 1.1.1 |
|--|------------------|
| <p>A large collection of organized data.</p> <ul style="list-style-type: none">• Can apply to any structured collection of data (e.g a relational table, data structures such as vectors & sets, graphs etc.) | |
| System | Definition 1.1.2 |
| <p>A collection of components interacting to achieve a greater goal.</p> <ul style="list-style-type: none">• 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) <p>Large & complex systems are typically developed over years by multiple teams.</p> | |
| Data Management System | Definition 1.1.3 |
| <p>A system built to control the entire lifecycle of some data.</p> <ul style="list-style-type: none">• Creation, modification, inspection and deletion of data• Classic examples include <i>Database Management Systems</i> | |
| Data Processing System | Definition 1.1.4 |
| <p>A system for processing data.</p> <ul style="list-style-type: none">• Support part of the data lifecycle• A strict superset of Data Management Systems (all data management systems are data processing systems) <p>For example a tool as small as grep could be considered a data processing system.</p> | |

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) |

1.2 Data Intensive Applications

| Data Intensive Application | Definition 1.2.1 |
|--|------------------|
| 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 1.2.1 |
|--|------------------------|
| Provide some examples of <i>Reporting</i> pattern being used in industry. | |
| <ul style="list-style-type: none"> • 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 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):

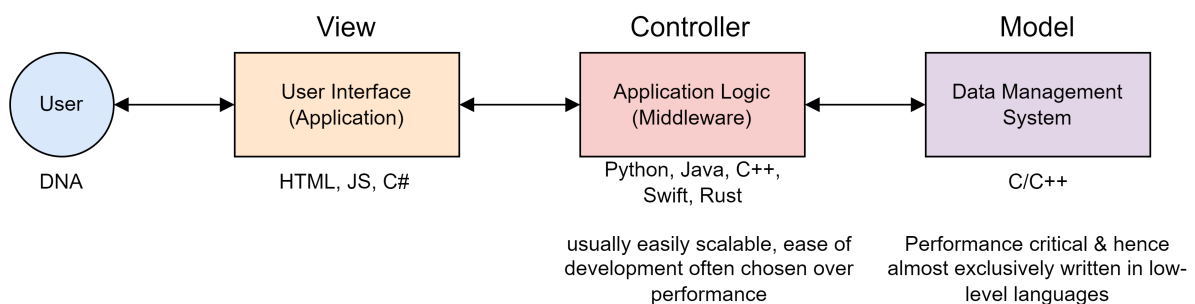
- Applications are domain-specific, and hence contain domain-specific optimisations that prevent fully general-purpose usage
- Data Management Systems are required to be highly generalised
- The cost of application specific data management (e.g developer time) outweighs any benefits for the majority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

Model View Controller (MVC)

Definition 1.2.2

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! 1.2.2

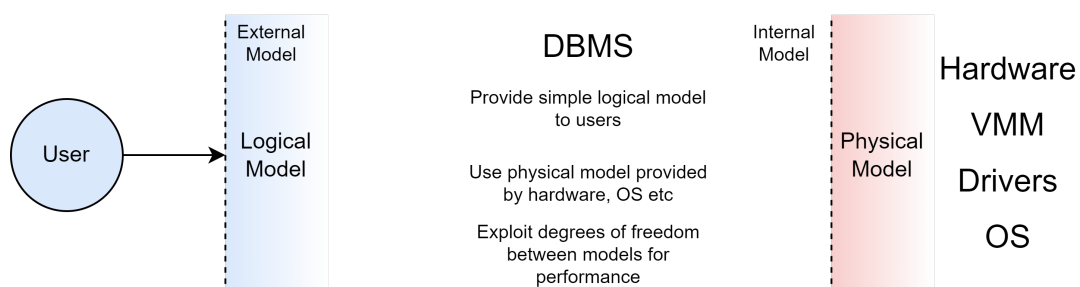
The enterprise data management systems market has been valued at \$82.25 billion (2021) with annual growth exceeding 10% (grand view research).

1.3 Data Management Systems

1.3.1 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. |

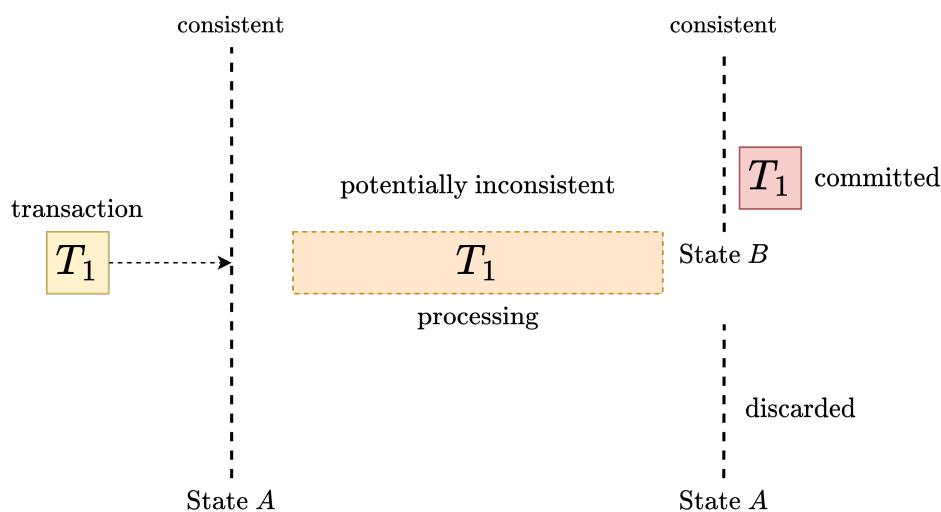
1.3.2 Logical/Physical Data Model Separation



1.3.3 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 1.3.1 |
|---|--|
| 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! 1.3.1 |
|---|---|
| In order to support efficient concurrent access & mutation of data without race conditions concurrency control is used: | |
| Lock Based | Each object (e.g record, table) contains a lock (read-write) used for synchronisation of access. The most common technique is <i>two-phase locking</i> . |
| Multiversion | Each object and transaction is timestamped, by maintaining multiple timestamped versions of an object a transaction can effectively operate on a snapshot of the database at its own timestamp. |

1.3.4 Read Phenomena

| Dirty Read / Uncommitted Dependency | Definition 1.3.2 |
|---|------------------|
| A transaction reads a record updated by a transaction that has not yet committed. | |
| <ul style="list-style-type: none"> The uncommitted transaction may fail or be rolled back rendering the dirty-read data invalid. | |

| Non-Repeatable Read | Definition 1.3.3 | Phantom Reads | Definition 1.3.4 |
|---|------------------|--|------------------|
| When a transaction reads a record twice with different results (another committed transaction updated the row between the reads). | | When a transaction reads a set of records twice, but the sets of records are not equal as another transaction committed between the reads. | |

1.3.5 Isolation levels

| Serialisable | Definition 1.3.5 | | |
|---|--------------------------------|---|----------------------------------|
| | <i>Dirty Read</i> Prevented | <i>Non-repeatable Read</i> Prevented | <i>Phantom Read</i> Prevented |
| Execution of transactions is can be serialized (it is equivalent to some sequential history of transactions). | | | |
| <ul style="list-style-type: none"> In lock-based concurrency control locks are released at the end of a transaction, and range-locks are acquired for <code>SELECT ... FROM ... WHERE ...</code> ; to avoid <i>phantom reads</i>. Prevents all 3 read phenomena and is the strongest isolation level. | | | |

| Repeatable Reads | Definition 1.3.6 | | |
|---|--------------------------------|---|--------------------------------|
| | <i>Dirty Read</i> Prevented | <i>Non-repeatable Read</i> Prevented | <i>Phantom Read</i> Allowed |
| <ul style="list-style-type: none"> Unlike <i>serialisable</i> Range locks are not used, only locks per-record. Write skew can occur (when concurrent transactions write to the same table & column using data read from the table, resulting in a mix of both transactions) | | | |

| Read Committed | Definition 1.3.7 | | |
|---|--------------------------------|---------------------------------------|--------------------------------|
| | <i>Dirty Read</i> Prevented | <i>Non-repeatable Read</i> Allowed | <i>Phantom Read</i> Allowed |
| Mutual exclusion is held for writes, but reads are only exclusive until the end of a <code>SELECT ...</code> ; statement, not until commit time. | | | |
| <ul style="list-style-type: none"> In lock-based concurrency, write locks are held until commit, read locks released after select completed. | | | |

| Read Uncommitted | Definition 1.3.8 | | |
|--|------------------------------|---------------------------------------|--------------------------------|
| | <i>Dirty Read</i> Allowed | <i>Non-repeatable Read</i> Allowed | <i>Phantom Read</i> Allowed |
| The weakest isolation level and allows for all <i>read phenomena</i> . | | | |

1.3.6 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)

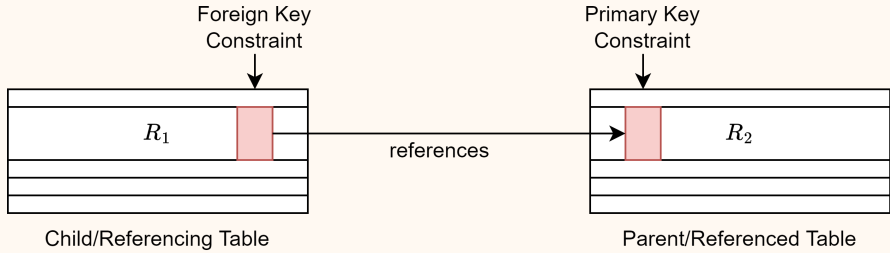
Chapter 2

Relational Algebra

2.1 Relational Structures

2.1.1 Preliminaries

| Schema | Definition 2.1.1 |
|---|------------------|
| A description of the database structure. | |
| <ul style="list-style-type: none">• Tables, names and types. | |
| <pre>CREATE TABLE foo (bing INTEGER, zog TEXT, bar INTEGER);</pre> | |
| <ul style="list-style-type: none">• Integrity constraints (foreign keys, nullability, uniqueness etc) | |
| <pre>ALTER TABLE foo ADD CONSTRAINT foo_key UNIQUE(bing);</pre> | |

| Foreign Key | Example Question 2.1.1 |
|--|------------------------|
| What is a foreign key constraint? Is it <i>like a pointer</i> ? | |
|  | |
| It adds the invariant that there is a record referenced by the foreign key. | |
| It is not really <i>like a pointer</i> as: | |
| <ul style="list-style-type: none">• 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 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|--|----------|----------|-----|------|---------|-------|---------|-------|---------|-------|---------|-------|---------|-------|---------|-------|---------|-------|---------|-------|----------|----------|----------|----------|---------|-------|---------|-------|
| <p>An array representing an n-ary relation R with the properties:</p> <ol style="list-style-type: none"> 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 <p>They are almost equivalent to sets tuples (but include labels).</p> | <div> <div>Columns:</div> <table> <tr> <th>X-ray</th> <th>Yankee</th> <th>...</th> <th>Zulu</th> </tr> <tr> <td>(x_1)</td> <td>y_1</td> <td>\dots</td> <td>z_1</td> </tr> <tr> <td>(x_2)</td> <td>y_2</td> <td>\dots</td> <td>z_2</td> </tr> <tr> <td>(x_3)</td> <td>y_3</td> <td>\dots</td> <td>z_3</td> </tr> <tr> <td>(x_4)</td> <td>y_4</td> <td>\dots</td> <td>z_4</td> </tr> <tr> <td>\vdots</td> <td>\vdots</td> <td>\vdots</td> <td>\vdots</td> </tr> <tr> <td>(x_n)</td> <td>y_n</td> <td>\dots</td> <td>z_n</td> </tr> </table> <div>Unique</div> </div> <div>Type (X, Y, \dots, Z)</div> | X-ray | Yankee | ... | Zulu | (x_1) | y_1 | \dots | z_1 | (x_2) | y_2 | \dots | z_2 | (x_3) | y_3 | \dots | z_3 | (x_4) | y_4 | \dots | z_4 | \vdots | \vdots | \vdots | \vdots | (x_n) | y_n | \dots | z_n |
| X-ray | Yankee | ... | Zulu | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (x_1) | y_1 | \dots | z_1 | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (x_2) | y_2 | \dots | z_2 | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (x_3) | y_3 | \dots | z_3 | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (x_4) | y_4 | \dots | z_4 | | | | | | | | | | | | | | | | | | | | | | | | | | |
| \vdots | \vdots | \vdots | \vdots | | | | | | | | | | | | | | | | | | | | | | | | | | |
| (x_n) | y_n | \dots | z_n | | | | | | | | | | | | | | | | | | | | | | | | | | |

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 |
|---|--|
| <p>Given the below structure, write a query.</p> <hr/> <pre> SELECT Book.title FROM ((Customer NATURAL JOIN Order) NATURAL JOIN OrderedItem) NATURAL JOIN Book </pre> <p>Here a natural join is:</p> $\text{natural join}(R_1, R_2) \triangleq \sigma_{R_1.x_1=R_2.x_1 \wedge \dots R_1.x_n=R_2.x_n}(R_1 \times R_2) \text{ where the } x\text{s are in both tables}$ | <p>UNFINISHED!!!</p> <p>$\Pi_{\text{title}}(\sigma_{\text{OrderItem.BookID}=\text{Book.BookID}}(\sigma_{\text{OrderedItem.OrderID} = \text{Order.OrderID}}((\sigma_{\text{Order.CustomerID}=\text{Customer.CustomerID}}(\sigma_{\text{customerID}=\text{Customer.CustomerID}}(\text{Customer} \times \text{Order}))))))$</p> |

| Unique Addresses | Example Question 2.1.3 |
|---|---|
| <p>UNFINISHED!!!</p> <pre> SELECT Book.Author FROM (SELECT Book.Author, Customer.ShippingAddress FROM ((Customer NATURAL JOIN Order) NATURAL JOIN OrderedItem) NATURAL JOIN Book) </pre> | <p>$\Pi_{\text{Book.Author}}(\sigma_{\text{count}=1}(\Gamma_{(\text{Customer.ShippingAddress}), (\text{Book.Author}, \text{count})}(\Pi_{\text{Book.Author}, \text{Customer.ShippingAddress}}(\text{natural join}(\text{Customer}, \text{Order}))))$</p> |

```

    ) 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).

```

#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...> {};
```

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

```

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

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

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

```

2.2.1 Relation

```

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

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

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

```

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

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

2.2.2 Project

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

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>;
```

```

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

2.2.4 Cross Product / Cartesian

$$R_1 \times R_2$$

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

```

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

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

```

2.2.5 Union

$$R_1 \cup R_2$$

The union of both relations, duplicates are eliminated.

```

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

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

```

2.2.6 Difference

$$R_1 - R_2$$

Get the set difference between two relations.

```

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

    LeftInputOperator leftInput;
    RightInputOperator rightInput;

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

```

2.2.7 Group Aggregation

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

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

This is implemented by **GROUP BY** in SQL:

```

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

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

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

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

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

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

```

2.2.8 Top-N

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

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

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

```

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

// note that here we include N in the type (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){};
};
```