VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

DATABASE MANAGEMENT SYSTEMS

PostgreSQL and HBase

Instructor: Assoc. Prof. Vo Thi Ngoc Chau, PhD

Students:

Nguyen Dinh Khuong Duy	1952207
Le Huy Hoang	1752209
Tran Tien Phat	1952386
Nguyen Vo Hoang Thi	1952996

HO CHI MINH CITY, MAY 31, 2023

Contents

1 Introduction

This is the report for the assignment of Database Management System (CO3021) of group 8 in semester HK221. In this report, we will present two databases, **PostgreSQL** and **Hbase**.

1.1 Requirements

The two main requirements of our group are:

- **Indexes**
- **Query Processing and Optimization**

In addition, our group also completed an extra requirement:

- **Data Storage and File Sturctures**

1.2 Report Structure

Our report structure is broken down as follows:

1. **Introduction**
   About our assignments and report.
2. **PostgreSQL**
   Study, research for the essential PostgreSQL documentation, and apply examples into practice.
3. **Hbase**
   Study, research for the essential HBase documentation, and apply examples into practice.
4. **Application**
   Introduce our application.
5. **Conclusion**
   Summarize for each database and compare according to the requirements assigned between the 2 databases. In addition, there are some examples to illustrate as required.
6. **Summary and Self's Assessment**
   Summary of our works and self-assessment.
7. **Workload**
   Details of each member's workload.
8. **References**
   List references to the resources and materials that we used to accomplish this assignment.

1.3 Submission

In our submission, we have attached three files:

- `DBMS_G8_1952207_1752209_1952386_1952996_Main_Report.pdf`
  Our main report.

- DBMS_G8_1952207_1752209_1952386_1952996_Progress_Time_Report.pdf
  Our extra progress time report.
- DBMS_G8_1952207_1752209_1952386_1952996_Application_Demonstration.mp4
  Our extra application demonstration.

The three files are combined into one zip file called DBMS_G8_1952207_1752209_1952386_1952996_Assignment.zip.

Please feel free to contact with us via this email if there are any missing, damaged, unavailable or unreadable files, or any additional issues.

2 PostgreSQL

In this section, we will cover and present the required contents related to PostgreSQL database.

2.1 Introduction to PostgreSQL

- POSTGRES, specifically Version 4.2, created at the University of California at Berkeley Computer Science Department, is the foundation for PostgreSQL, an object-relational database management system (ORDBMS). Many ideas that POSTGRES pioneered were later made available in certain commercial database systems.
- In this assignment, we are using PostgreSQL version 14 and pgAdmin 4.
- Most of the information, resources and materials for PostgreSQL are located [22].
- Beside that, we rely on some database knowledge written in [5, 20, 24].
- PostgreSQL is an open-source descendant of this original Berkeley code. It supports a large part of the SQL standard and offers many modern features:
  - Complex queries.
  - Foreign keys.
  - Triggers .
  - Updatable views.
  - Transactional integrity.
  - Multiversion concurrency control.
- Also, PostgreSQL can be extended by the user in many ways, for example by adding new
  - Data types.
  - Functions.
  - Operators.
  - Aggregate functions.
  - Index methods.
  - Procedural languages.
- And because of the liberal license, PostgreSQL can be used, modified, and distributed by anyone free of charge for any purpose, be it private, commercial, or academic.

2.2 Basic PostgreSQL Commands

We need to be familiar with a few PostgreSQL commands in addition to the standard SQL queries.

- Login.

```
psql -U postgres
```

```
postgres-#
```

- List all user.

```
postgres-# \du
```

```
 Role name |                         Attributes                         |
 Member of
-----------+------------------------------------------------------------+--------
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
```

- List all database.

```
postgres-# \l
```

- Access specific to database `database_name`.

```
postgres=# \c database_name
```

- List all table.

```
database_name=# \d
```

- Import `data.sql` data file to `database_name` database.

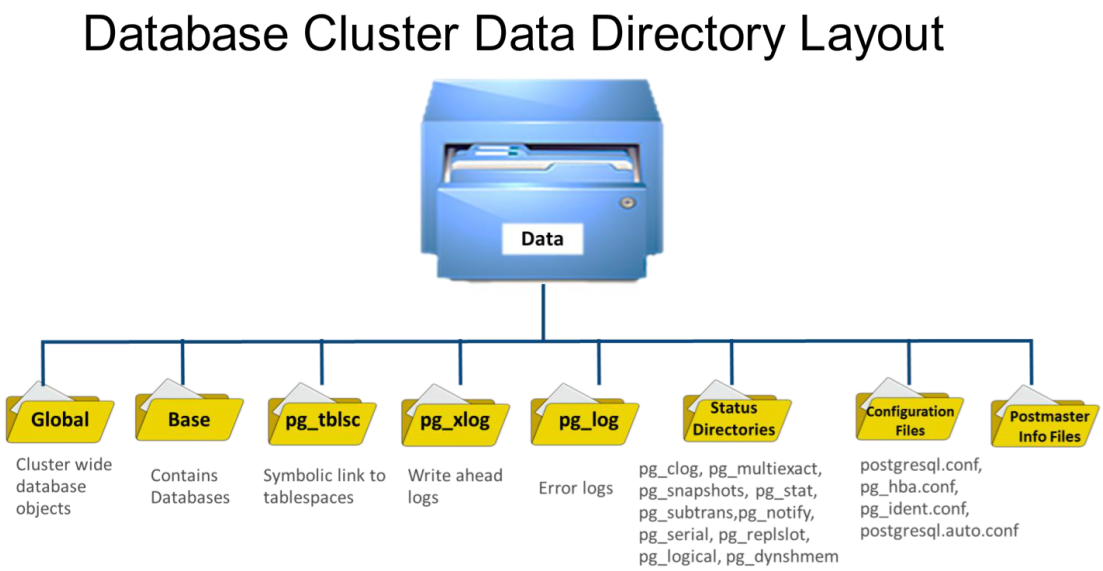
```
psql -h localhost -U postgres -d database_name -f ./data.sql
```

2.3 Data Storage and File Structures

The physical structure of PostgreSQL is straightforward; it consists of the following components:

- Shared memory
- Background processes
- Data directory structure / Data files

2.3.1 Data Directory



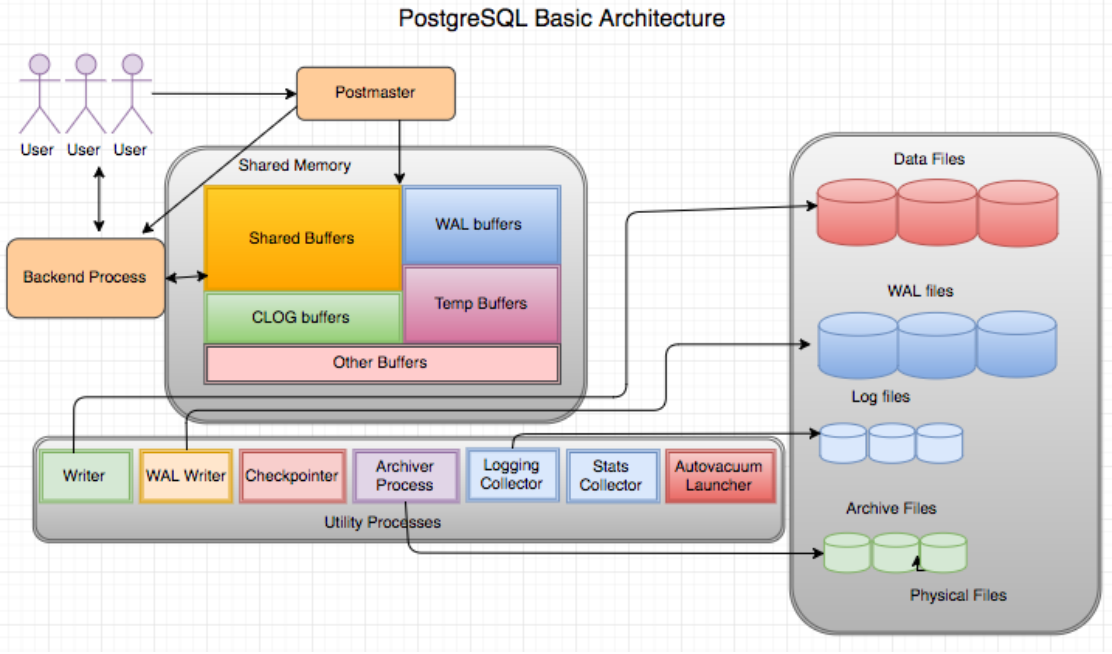
A database cluster's configuration and data files are typically kept together in the cluster's data directory, also known as `PGDATA` (after the name of the environ- ment variable that can be used to define it). A common location for `PGDATA` is `/var/lib/pgsql/data`. Multiple clusters, managed by different server instances, can exist on the same machine.

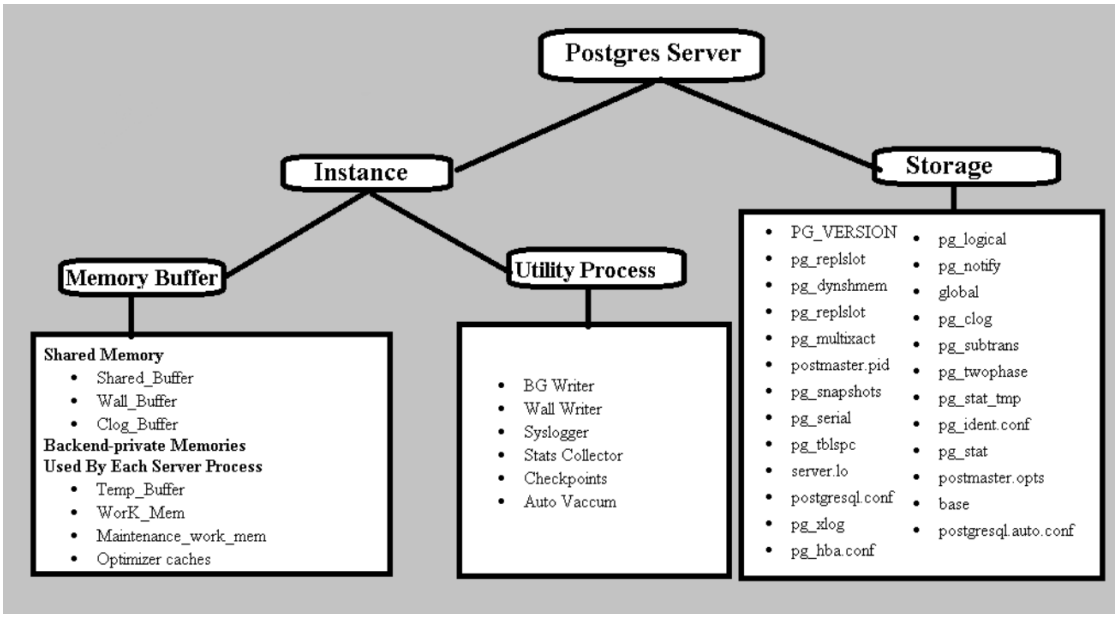
Specifies the directory to use for data storage. This parameter can only be set at server start.

- `config_file`: Specifies the main server configuration file called `postgresql.conf`. This parameter can only be set on the PostgreSQL command line.
- `hba_file`: Specifies the configuration file for host-based authentication called `pg_hba.conf`. This parameter can only be set at the server start.
- `ident_file`: Specifies the configuration file for user name mapping called `pg_ident.conf`. This parameter can only be set at the server start.
- `postmaster.opts file`: A file recording the command-line options the postmaster was last started with
- `postmaster.pid file`: A lock file recording the current postmaster PID and shared memory segment ID (not present after postmaster shutdown)
- `pg_version file` : A file containing the major version number of PostgreSQL.
- `pg_wal directory`: Here the write-ahead logs are stored. It is the log file, where all the logs are stored of committed and uncommitted transactions. It contains a maximum of 6 logs, and the last one overwrites. If the archiver is on, it moves there.
- `base directory`: Subdirectory containing per-database subdirectories.

- `global directory`: Subdirectory containing cluster-wide tables, such as `pg_database`.
- `pg_multixact directory`: Subdirectory containing multi transaction status data (used for shared row locks).
- `pg_subtrans directory`: Subdirectory containing subtransaction status data.
- `pg_tblspc directory`: Subdirectory containing symbolic links to tablespaces.
- `pg_twophase directory`: Subdirectory containing state files for prepared transactions.

2.3.2 Physical Files





- `Data Files`: It contains all the data required by the user for processing. Data from data files are first moved to shared buffers and then the data is processed accordingly.
- `WAL Files/Segments`: WAL Buffers are linked with WAL Files/Segments. WAL buffer flushes all the data into WAL Segment whenever commit command occurs. Commit is a command which ensures the end of the transaction. Commit work by default after every command in PostgreSQL which can also be changed accordingly.
- `Archived WAL/Files`: This file segment is a kind of copy of WAL Segment which stores all the archived data of WAL Segment which can be stored for a long period. WAL segment is having limited storage capacity. Once space is exhausted in WAL Segment, the system starts replacing the old data with new data which results in the loss of old data. Archives are required till full backup of the database. After that, we can even delete the archive in order to gain some space in the DB.
- `Error Log Files`: Those files which contain all the error messages, warning messages, informational messages, or messages belonging to all the major thing happening to the database. All logs related to DDL and DML changes are not stored in this space. Internal errors, internal bugs entry is stored in this file which helps the admin to troubleshoot the problem.
- `shared_buffer`: Sets the amount of memory the database server uses for shared memory buffers. The default is typically 128 megabytes (128MB), but might be less if your kernel settings will not support it (as determined during `initdb`). This setting must be at least 128 kilobytes. (Non-default values of `BLCKSZ` change the minimum.) However, settings significantly higher than the minimum are usually needed for good performance. This parameter can only be set at server start.

  If you have a dedicated database server with 1GB or more of RAM, a reasonable starting value for `shared_buffers` is 25% of the memory in your system. There are some workloads

where even large settings for shared_buffers are effective, but because PostgreSQL also relies on the operating system cache, it is unlikely that an allocation of more than 40% of RAM to `shared_buffers` will work better than a smaller amount. Larger settings for `shared_buffers` usually require a corresponding increase in `checkpoint_segments`, in order to spread out the process of writing large quantities of new or changed data over a longer period of time.

On systems with less than 1GB of RAM, a smaller percentage of RAM is appropriate, so as to leave adequate space for the operating system. Also, on Windows, large values for `shared_buffer` are not as effective. You may find better results keeping the setting relatively low and using the operating system cache more instead. The useful range for `shared_buffer` on Windows systems is generally from 64MB to 512MB.

- `temp_buffers`: Sets the maximum number of temporary buffers used by each database session.

  These are session-local buffers used only for access to temporary tables. The default is eight megabytes (8MB). The setting can be changed within individual sessions, but only before the first use of temporary tables within the session; subsequent attempts to change the value will have no effect on that session.

  A session will allocate temporary buffers as needed up to the limit given by `temp_buffers`. The cost of setting a large value in sessions that do not actually need many temporary buffers is only a buffer descriptor, or about 64 bytes, per increment in `temp_buffers`. However if a buffer is actually used an additional 8192 bytes will be consumed for it (or in general, `BLCKSZ` bytes).

- `Clog Buffers`: Clog buffers are one of the SLRU-style buffers oriented toward circular "rings" of data, like which transaction numbers have been committed or rolled back.

- `work_mem`: Specifies the amount of memory to be used by internal sort operations and hash tables before writing to temporary disk files. The value defaults to four megabytes (4MB).

  Note that for a complex query, several sort or hash operations might be running in parallel; each operation will be allowed to use as much memory as this value specifies before it starts to write data into temporary files.

  Also, several running sessions could be doing such operations concurrently. Therefore, the total memory used could be many times the value of `work_mem`; it is necessary to keep this fact in mind when choosing the value. Sort operations are used for `ORDER BY`, `DISTINCT`, and merge joins.

  Hash tables are used in hash joins, hash-based aggregation, and hash-based processing of `IN` subqueries.

- `maintenance_work_mem`: Specifies the maximum amount of memory to be used by maintenance operations, such as `VACUUM`, `CREATE INDEX`, and `ALTER TABLE ADD FOREIGN KEY`. It defaults to 64 megabytes (64MB).

  Since only one of these operations can be executed at a time by a database session, and an installation normally does not have many of them running concurrently, it is safe to set this value significantly larger than `work_mem`. Larger settings might improve performance for vacuuming and for restoring database dumps.

  Note that when `autovacuum` runs, up to `autovacuum_max_workers` times this memory may be allocated, so be careful not to set the default value too high. It may be useful to control for this by separately setting `autovacuum_work_mem`.

- `BGWriter`: There is a separate server process called the background writer, whose function is to issue writes of new or modified shared buffers.

  It writes shared buffers so server processes handling user queries seldom or never need to wait for a write to occur.

  However, the background writer does cause a net overall increase in I/O load, because while a repeatedly-dirtied page might otherwise be written only once per checkpoint interval, the background writer might write it several times as it is dirtied in the same interval.

  The parameters discussed in this subsection can be used to tune the behavior for local needs.

- `Wal_Writer`: `WAL_buffers` are written out to disk at every transaction commit, so extremely large values are unlikely to provide a significant benefit. However, setting this value to at least a few megabytes can improve write performance on a busy server where many clients are committing at once. The auto-tuning selected by the default setting of -1 should give reasonable results in most cases.

  Specifies the delay between activity rounds for the `WAL_writer`. In each round the writer will flush WAL to disk. It then sleeps for `wal_writer_delay` milliseconds, and repeats.

  The default value is 200 milliseconds (200ms). Note that on many systems, the effective resolution of sleep delays is 10 milliseconds; setting `wal_writer_delay` to a value that is not a multiple of 10 might have the same results as setting it to the next higher multiple of 10. This parameter can only be set in the `postgresql.conf` file or on the server command line [27].

- `SysLogger` - Error Reporting and Logging

  As per the figure, it is clearly understood that all – the utility processes + user backends + Postmaster Daemon are attached to `syslogger` process for logging the information about their activities. Every process information is logged under `\$PGDATA/pg_log` with the `file.log`.

  Debugging more on the process information will cause overhead on the server. Minimal tuning is always recommended. However, increasing the debug level when required.

  `logging collector`: this is a background process that captures log messages sent to stderr and redirects them into log files

  `log_directory`- Data directory

  `log_filename` - The default is `postgresql-\Y-\m-\d\_\H\M\S.log` and the default permissions are 0600.

- `Checkpoints`: checkpoint will be occurs following scenarios:

  1. `pg_start_backup`.
  2. `CREATE DATABASE`.
  3. `pg_ctl stop/restart`.
  4. `pg_stop_backup`.
  5. Issue of commit.
  6. Pages are dirty.
  7. And a few others.

2.3.3 Data File Layout (OID)

All database objects in PostgreSQL are managed internally by their respective object identifiers (OIDs), which are assigned 4 byte integers. The OID of the database is stored in the `pg_database` system table. The OIDs of objects such as tables, indexes, and sequences in the database are stored in the `pg_class` system table.

In the below example, we derived OID of database, Tablespace & Table and we can relate all OIDs with function `pg_relation_filepath`, this will give you the exact location where table datafiles located.

```
postgres=# \c dvdrental
You are now connected to database "dvdrental" as user "postgres".
dvdrental=# select oid,datname,datdba,dattablespace from pg_database where datname='dvdrental';
  oid  |  datname  | datdba | dattablespace
-------+-----------+--------+---------------
 16468 | dvdrental |     10 |          1663
(1 row)


dvdrental=# select oid,spcname,spcowner from pg_tablespace where spcname='myts01';
 oid | spcname | spcowner
-----+---------+----------
(0 rows)


dvdrental=# SELECT oid,relname,relkind FROM pg_class WHERE relname = 'address';
  oid  | relname | relkind
-------+---------+---------
 16540 | address | r
(1 row)


dvdrental=# SELECT pg_relation_filepath('address');
 pg_relation_filepath
----------------------
 base/16468/16540
(1 row)


dvdrental=#
```

When data is stored in PostgreSQL, PostgreSQL in turn stores that data in regular files in the filesystem.Each table in PostgreSQL is represented by one or more underlying files. Each 1GB chunk of the table is stored in a separate file. It is actually pretty easy to find the actual underlying files for a table. To do so, you first need to find the PostgreSQL data directory, which is the directory in which PostgreSQL keeps all of your data. You can find where the data directory is by running `SHOW DATA_DIRECTORY`.

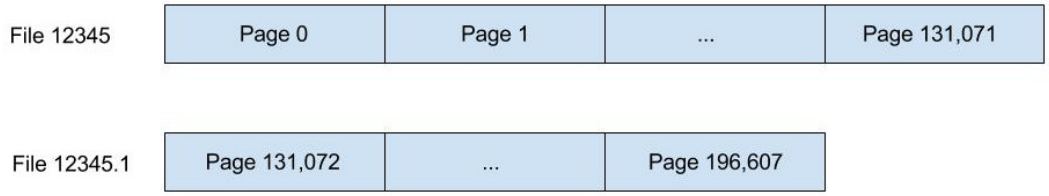
```
postgres=# SHOW DATA_DIRECTORY;
           data_directory
-----------------------------------------
 C:/Program Files/PostgreSQL/14/data
(1 row)


postgres=#
```

Now that knowing where the PostgreSQL data directory is, the need to find where the files for the specific table we are looking for is located. To do so, using the `pg_relation_filepath` function with the name of the table you want to find the file for. The function will return the relative `filepath` of the files from the data directory.

```
test=# SELECT pg_relation_filepath('car');
 pg_relation_filepath
----------------------
 base/16394/16455
(1 row)
```

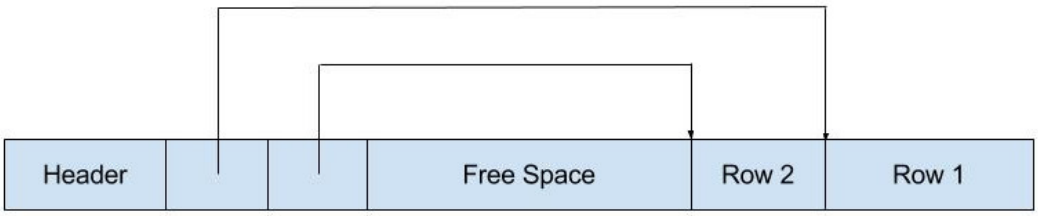
Together with the location of the location of the data directory, this gives us the location of the files for the people table. All of the files are stored in `/var/lib/postgresql/14/main/base/16394/`. The first GB of the table is stored in a file called 16455, the second in a file called 16455.1, the third in 16455.2, and so on.

How each file is laid out. Each file is broken up into 8 kB chunks, called “pages”. For example, a 1.5GB table will be stored across two files and 196,608 pages and look like the following:



Each row is stored on a single page (with the exception of when a row is too large, in which case a technique called TOAST is used). Pages are the unit of which PostgreSQL reads and writes data to the filesystem. Whenever PostgreSQL reads a row it needs to answer a query from disk , PostgreSQL reads the entire page the row is on. When PostgreSQL writes to a row on a page, it writes a whole new copy of the entire page to disk at one time. Pages themselves have the following format:

The header is 24 bytes and contains various metadata about the page, including a checksum and information necessary for WAL. The row offsets is of pointers into the rows field, with the N_{th} pointer pointing to the N_{th} row. The offsets can be used to quickly lookup an arbitrary row of a page.



The first thing which is likely to be noticed is that the first rows are stored at the back of the page. That is so the offsets and the actual row data can both grow towards the middle. If a new row is inserted, we can allocate a new offset from the front of the free space, and allocate the space for the row from the back of the free space. As for each row, they have a format that looks like the following:

The header of each row is 23 bytes and includes the transaction ids for MVCC as well as other metadata about the row. Based on the table schema, each field of the row is either a fixed width type or a variable width type. If the field is fixed width, PostgreSQL already knows how long the field is and just stores the field data directly in the row.

If the field is variable width there are two possibilities for how the field is stored. Under normal circumstances, it would be stored directly the row with a header detailing how large the field is. In certain special cases, or when it is impossible to store the field directly in the row, the field will be stored outside of the row using a technique using TOAST (The Oversized-Attribute Storage Technique).

2.3.4 Database Structure

Here are some things that are important to know when attempting to understand the database structure of **PostgreSQL**.

Items related to the database

1. PostgreSQL consists of several databases. This is called a database cluster.
2. When `initdb()` is executed, `template0` , `template1` , and `postgres` databases are created.
3. The `template0` and `template1` databases are template databases for user database creation and contain the system catalog tables.
4. The list of tables in the `template0` and `template1` databases is the same immediately after `initdb()`. However, the template1 database can create objects that the user needs.
5. The user database is created by cloning the `template1` database.

Items related to the tablespace

1. The `pg_default` and `pg_global` tablespaces are created immediately after `initdb()`.
2. If you do not specify a tablespace at the time of table creation, it is stored in the `pg_default` tablespace.
3. ables managed at the database cluster level are stored in the `pg_global` tablespace.
4. The physical location of the `pg_default` tablespace is `$PGDATA/base`.
5. The physical location of the `pg_global` tablespace is `$PGDATA/global`.
6. One tablespace can be used by multiple databases. At this time, a database-specific subdirectory is created in the tablespace directory.

7. Creating a user tablespace creates a symbolic link to the user tablespace in the `$PGDATA/tblspc` directory.

Items related to the table

1. There are three files per table.
2. One is a file for storing table data. The file name is the OID of the table.
3. One is a file to manage table-free space. The file name is `OID_fsm`.
4. One is a file for managing the visibility of the table block. The file name is `OID_vm`.
5. The index does not have a `_vm` file. That is, OID and `OID_fsm` are composed of two files.

```
postgres=# \l
                                  List of databases
   Name    |  Owner   | Encoding |   Collate   |    Ctype    |   Access privileges
-----------+----------+----------+-------------+-------------+-----------------------
 postgres  | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 template0 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres          +
           |          |          |             |             | postgres=CTc/postgres
 template1 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres          +
           |          |          |             |             | postgres=CTc/postgres

postgres=# \db
          List of tablespaces
    Name    |  Owner   |    Location
------------+----------+-----------------
 pg_default | postgres |
 pg_global  | postgres |

[root@test-machine02 17115]# pwd
/var/lib/pgsql/13/data/base/17115
[root@test-machine02 17115]#
[root@test-machine02 17115]# ls -ltr *17246*
-rw-------. 1 postgres postgres   8192 May 20 16:46 17246_vm
-rw-------. 1 postgres postgres  24576 May 20 16:46 17246_fsm
-rw-------. 1 postgres postgres 884736 May 20 16:46 17246
[root@test-machine02 17115]#
[root@test-machine02 17115]#

postgres=#
postgres=# select oid,datname,datdba,dattablespace from pg_database where oid=17115;
  oid  | datname | datdba | dattablespace
-------+---------+--------+---------------
 17115 | db1     |     10 |          1663
(1 row)

postgres=# \connect db1
You are now connected to database "db1" as user "postgres".
db1=# SELECT oid,relname,relkind FROM pg_class WHERE oid=17246;
  oid  | relname | relkind
-------+---------+---------
 17246 | payment | r
(1 row)

db1=# SELECT pg_relation_filepath('payment');
 pg_relation_filepath
----------------------
 base/17115/17246
(1 row)

db1=#

[root@test-machine02 pg_tblspc]#
[root@test-machine02 pg_tblspc]# pwd
/var/lib/pgsql/13/data/pg_tblspc
[root@test-machine02 pg_tblspc]#
[root@test-machine02 pg_tblspc]# ls -ltr
total 0
lrwxrwxrwx. 1 postgres postgres 14 May 25 14:03 23701 -> /u01/psql_tbls
lrwxrwxrwx. 1 postgres postgres 15 May 25 15:02 23704 -> /u01/psql2_tbls
[root@test-machine02 pg_tblspc]#
```

2.4 Index types in PostgreSQL

- An index is a **special lookup table** that the Database Search Engine can use to speed up the performance time and to be efficiency. To simplify, an index is a pointer to data in a table. An index in a Database is similar as an index in the book's Table of Contents.

- User can not see the indexes, they are used to improves the speed of data retrieval operations and quickly locate data without having to search every row in a database table every time a database table is accessed.
- The `INSERT` and `UPDATE` command will take more time to be finished but the `SELECT` command will become faster to process. The reason is when update and insert data into the database, indexes for them will be inserted or updated also.
- PostgreSQL provides several index types: B-tree, Hash, GiST, SP-GiST, GIN and BRIN. Each index type uses a different algorithm that is best suited to different types of queries [13, 14].
- By default, the `CREATE INDEX` command creates B-tree indexes, which fit the most common situations. The other index types are selected by writing the keyword `USING` followed by the index type name.
- For example, to create a Hash index:

```
1 CREATE INDEX name ON table USING HASH (column);
```

2.4.1 Create Index

We can create index in PostgreSQL with this command:

```
1 CREATE INDEX index_name ON table_name USING [method_name];
```

```
test=# CREATE INDEX price_index ON car USING BRIN(price);
CREATE INDEX
test=# \d+ car
                                        Table "public.car"
 Column |          Type          | Collation | Nullable | Default | Storage  | Compression | Stats target | Description
--------+------------------------+-----------+----------+---------+----------+-------------+--------------+-------------
 id     | integer                |           | not null |         | plain    |             |              |
 make   | character varying(100) |           | not null |         | extended |             |              |
 model  | character varying(100) |           | not null |         | extended |             |              |
 year   | integer                |           |          |         | plain    |             |              |
 price  | numeric(19,2)          |           | not null |         | main     |             |              |
Indexes:
    "price_index" brin (price)
Access method: heap
```

2.4.2 Delete Index

We can delete index in PostgreSQL with this command:

```
1 DROP INDEX index_name;
```

```
test=# \d+ car
                                                Table "public.car"
 Column |          Type          | Collation | Nullable | Default | Storage  | Compression | Stats target | Description
--------+------------------------+-----------+----------+---------+----------+-------------+--------------+-------------
 id     | integer                |           | not null |         | plain    |             |              |
 make   | character varying(100) |           | not null |         | extended |             |              |
 model  | character varying(100) |           | not null |         | extended |             |              |
 year   | integer                |           |          |         | plain    |             |              |
 price  | numeric(19,2)          |           | not null |         | main     |             |              |
Indexes:
    "make_index" hash (make)
Access method: heap


test=# DROP INDEX make_index;
DROP INDEX
test=# \d+ car
                                                Table "public.car"
 Column |          Type          | Collation | Nullable | Default | Storage  | Compression | Stats target | Description
--------+------------------------+-----------+----------+---------+----------+-------------+--------------+-------------
 id     | integer                |           | not null |         | plain    |             |              |
 make   | character varying(100) |           | not null |         | extended |             |              |
 model  | character varying(100) |           | not null |         | extended |             |              |
 year   | integer                |           |          |         | plain    |             |              |
 price  | numeric(19,2)          |           | not null |         | main     |             |              |
Access method: heap
```

2.4.3 B-tree

- B-trees can handle equality and range queries on data that can be sorted into some ordering. In particular, the PostgreSQL query planner will consider using a B-tree index [16] whenever an indexed column is involved in a comparison using one of these operators:

  $$< \quad <= \quad = \quad >= \quad >$$

- Constructs equivalent to combinations of these operators, such as `BETWEEN` and `IN`, can also be implemented with a B-tree index search. Also, an `IS NULL` or `IS NOT NULL` condition on an index column can be used with a B-tree index.

- The optimizer can also use a B-tree index for queries involving the pattern matching operators `LIKE` and `~ if` the pattern is a constant and is anchored to the beginning of the string — for example, col LIKE `'foo%'` or `col ~ '^foo'`, but not `col LIKE '%bar'`. It is also possible to use B-tree indexes for `ILIKE` and `~*`, but only if the pattern starts with non-alphabetic characters, i.e., characters that are not affected by upper/lower case conversion.

- B-tree indexes can also be used to retrieve data in sorted order. This is not always faster than a simple scan and sort, but it is often helpful.

```
public_car=# CREATE INDEX price_index ON car USING btree(price);
CREATE INDEX
public_car=# \d+ car
                                                Table "public.car"
 Column |          Type          | Collation | Nullable | Default | Storage  | Compression | Stats target | Description
--------+------------------------+-----------+----------+---------+----------+-------------+--------------+-------------
 id     | integer                |           | not null |         | plain    |             |              |
 make   | character varying(100) |           | not null |         | extended |             |              |
 model  | character varying(100) |           | not null |         | extended |             |              |
 year   | integer                |           |          |         | plain    |             |              |
 price  | numeric(19,2)          |           | not null |         | main     |             |              |
Indexes:
    "price_index" btree (price)
Access method: heap

public_car=#
```

```
1 CREATE INDEX price_index ON car USING btree(price);
```

```
1 EXPLAIN (ANALYZE, VERBOSE) SELECT * FROM car WHERE price > '40000';
```

```
public_car=# EXPLAIN (ANALYZE, VERBOSE) SELECT * FROM car WHERE price > '40000';
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on public.car  (cost=0.00..21.50 rows=579 width=30) (actual time=0.031..0.465 rows=578 loops=1)
   Output: id, make, model, year, price
   Filter: (car.price > '40000'::numeric)
   Rows Removed by Filter: 422
 Planning Time: 0.208 ms
 Execution Time: 0.548 ms
(6 rows)

public_car=#
```

2.4.4 Hash

- Hash indexes store a 32-bit hash code derived from the value of the indexed column. Hence, such indexes can only handle simple equality comparisons. The query planner will consider using a hash index [15] whenever an indexed column is involved in a comparison using the equal operator:

  ```
  =
  ```

```
1 CREATE INDEX index_name ON table_name USING hash(column_name);
```

- SELECT function without index:

```
test=# \d+ car
                                                Table "public.car"
 Column |          Type          | Collation | Nullable | Default | Storage  | Compression | Stats target | Description
--------+------------------------+-----------+----------+---------+----------+-------------+--------------+-------------
 id     | integer                |           | not null |         | plain    |             |              |
 make   | character varying(100) |           | not null |         | extended |             |              |
 model  | character varying(100) |           | not null |         | extended |             |              |
 year   | integer                |           |          |         | plain    |             |              |
 price  | numeric(19,2)          |           | not null |         | main     |             |              |
Access method: heap


test=# EXPLAIN(ANALYZE,VERBOSE) SELECT * FROM car WHERE make = 'Honda';
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Seq Scan on public.car  (cost=0.00..21.50 rows=28 width=30) (actual time=0.029..0.195 rows=28 loops=1)
   Output: id, make, model, year, price
   Filter: ((car.make)::text = 'Honda'::text)
   Rows Removed by Filter: 972
 Planning Time: 1.490 ms
 Execution Time: 0.221 ms
(6 rows)
```

- SELECT function with hash index:

```
test=# EXPLAIN(ANALYZE,VERBOSE) SELECT * FROM car WHERE make = 'Honda';
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on public.car  (cost=4.22..13.57 rows=28 width=30) (actual time=0.027..0.048 rows=28 loops=1)
   Output: id, make, model, year, price
   Recheck Cond: ((car.make)::text = 'Honda'::text)
   Heap Blocks: exact=8
   ->  Bitmap Index Scan on make_index  (cost=0.00..4.21 rows=28 width=0) (actual time=0.015..0.015 rows=28 loops=1)
         Index Cond: ((car.make)::text = 'Honda'::text)
 Planning Time: 0.130 ms
 Execution Time: 0.090 ms
(8 rows)
```

2.4.5 GiST

- GiST stands for **Generalized Search Tree**. It is a balanced, tree-structured access method, that acts as a base template in which to implement arbitrary indexing schemes. B-trees, R-trees and many other indexing schemes can be implemented in GiST [17].
- GiST indexes are not a single kind of index, but rather an infrastructure within which many different indexing strategies can be implemented. Accordingly, the particular operators with which a GiST index can be used vary depending on the indexing strategy (the operator class). As an example, the standard distribution of PostgreSQL includes GiST operator classes for several two-dimensional geometric data types, which support indexed queries using these operators:

  $$<<\quad \&<\quad \&>\quad >>\quad <<|\quad \&<|\quad |\&>\quad |>>\quad @>\quad <@\quad \approx=\quad \&\&$$

- Many other GiST operator classes are available in the `contrib` collection or as separate projects. GiST indexes are also capable of optimizing "nearest-neighbor" searches, such as:

```
1 SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

which finds the ten places closest to a given target point. The ability to do this is again dependent on the particular operator class being used.

- SELECT function without index:

```
test=# \d+ geocoordinate
                                    Table "public.geocoordinate"
   Column   | Type  | Collation | Nullable | Default | Storage | Compression | Stats target | Description
------------+-------+-----------+----------+---------+---------+-------------+--------------+-------------
 coordinate | point |           |          |         | plain   |             |              |
Access method: heap


test=# EXPLAIN(ANALYZE,VERBOSE) SELECT FROM geocoordinate ORDER BY coordinate <-> '(29.5,-20.6)';
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=844.39..869.39 rows=10000 width=8) (actual time=7.307..7.748 rows=10000 loops=1)
   Output: ((coordinate <-> '(29.5,-20.6)'::point))
   Sort Key: ((geocoordinate.coordinate <-> '(29.5,-20.6)'::point))
   Sort Method: quicksort  Memory: 853kB
   ->  Seq Scan on public.geocoordinate  (cost=0.00..180.00 rows=10000 width=8) (actual time=0.031..2.576 rows=10000 loops=1)
         Output: (coordinate <-> '(29.5,-20.6)'::point)
 Planning Time: 0.107 ms
 Execution Time: 8.304 ms
(8 rows)
```

- SELECT function with GiST index:

```
test=# CREATE INDEX idx_gist_coordinate ON geocoordinate USING gist(coordinate);
CREATE INDEX
test=# EXPLAIN(ANALYZE,VERBOSE) SELECT FROM geocoordinate ORDER BY coordinate <->'(29.5,-20.6)' LIMIT 5;
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=0.15..0.37 rows=5 width=8) (actual time=1.554..1.907 rows=5 loops=1)
   Output: ((coordinate <-> '(29.5,-20.6)'::point))
   ->  Index Only Scan using idx_gist_coordinate on public.geocoordinate  (cost=0.15..448.15 rows=10000 width=8) (actual time=1.552..1.903 rows=5 loops=1)
         Output: (coordinate <-> '(29.5,-20.6)'::point)
         Order By: (geocoordinate.coordinate <-> '(29.5,-20.6)'::point)
         Heap Fetches: 0
 Planning Time: 2.370 ms
 Execution Time: 1.936 ms
(8 rows)
```

2.4.6 SP-GiST

- SP-GiST is an abbreviation for **Space-Partitioned Generalized Search Tree**. SP-GiST supports partitioned search trees, which facilitate development of a wide range of different non-balanced data structures, such as quad-trees, k-d trees, and radix trees (tries). The common feature of these structures is that they repeatedly divide the search space into partitions that need not be of equal size. Searches that are well matched to the partitioning rule can be very fast [18].
- SP-GiST indexes, like GiST indexes, offer an infrastructure that supports various kinds of searches. SP-GiST permits implementation of a wide range of different non-balanced disk-based data structures, such as quadtrees, k-d trees, and radix trees (tries). As an example, the standard distribution of PostgreSQL includes SP-GiST operator classes for two-dimensional points, which support indexed queries using these operators:

<<	>>	≈=	<@	<<|	|>>

- Like GiST, SP-GiST supports “nearest-neighbor” searches.

```sql
CREATE INDEX index_name ON table_name USING spgist(column_name);
```

- SELECT function without index:

```
test=# \d+ geocoordinate
                                    Table "public.geocoordinate"
   Column    | Type  | Collation | Nullable | Default | Storage | Compression | Stats target | Description
-------------+-------+-----------+----------+---------+---------+-------------+--------------+-------------
 coordinate | point |           |          |         | plain   |             |              |
Access method: heap


test=# EXPLAIN(ANALYZE,VERBOSE) SELECT FROM geocoordinate ORDER BY coordinate <-> '(29.5,-20.6)';
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=844.39..869.39 rows=10000 width=8) (actual time=7.307..7.748 rows=10000 loops=1)
   Output: ((coordinate <-> '(29.5,-20.6)'::point))
   Sort Key: ((geocoordinate.coordinate <-> '(29.5,-20.6)'::point))
   Sort Method: quicksort  Memory: 853kB
   ->  Seq Scan on public.geocoordinate  (cost=0.00..180.00 rows=10000 width=8) (actual time=0.031..2.576 rows=10000 loops=1)
         Output: (coordinate <-> '(29.5,-20.6)'::point)
 Planning Time: 0.107 ms
 Execution Time: 8.304 ms
(8 rows)
```

- SELECT function with SP-GiST index:

```
test=# \d+ geocoordinate
                                    Table "public.geocoordinate"
   Column    | Type  | Collation | Nullable | Default | Storage | Compression | Stats target | Description
-------------+-------+-----------+----------+---------+---------+-------------+--------------+-------------
 coordinate | point |           |          |         | plain   |             |              |
Access method: heap


test=# EXPLAIN(ANALYZE,VERBOSE) SELECT FROM geocoordinate ORDER BY coordinate <-> '(29.5,-20.6)' LIMIT 5;
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=346.10..346.11 rows=5 width=8) (actual time=4.534..4.536 rows=5 loops=1)
   Output: ((coordinate <-> '(29.5,-20.6)'::point))
   ->  Sort  (cost=346.10..371.10 rows=10000 width=8) (actual time=4.532..4.533 rows=5 loops=1)
         Output: ((coordinate <-> '(29.5,-20.6)'::point))
         Sort Key: ((geocoordinate.coordinate <-> '(29.5,-20.6)'::point))
         Sort Method: top-N heapsort  Memory: 25kB
         ->  Seq Scan on public.geocoordinate  (cost=0.00..180.00 rows=10000 width=8) (actual time=0.027..2.746 rows=10000 loops=1)
               Output: (coordinate <-> '(29.5,-20.6)'::point)
 Planning Time: 0.108 ms
 Execution Time: 4.565 ms
(10 rows)


test=# CREATE INDEX idx_spgist_coordinate ON geocoordinate USING spgist(coordinate);
CREATE INDEX
test=# EXPLAIN(ANALYZE,VERBOSE) SELECT FROM geocoordinate ORDER BY coordinate <-> '(29.5,-20.6)' LIMIT 5;
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=0.15..0.37 rows=5 width=8) (actual time=0.769..0.779 rows=5 loops=1)
   Output: ((coordinate <-> '(29.5,-20.6)'::point))
   ->  Index Only Scan using idx_spgist_coordinate on public.geocoordinate  (cost=0.15..444.15 rows=10000 width=8) (actual time=0.767..0.776 rows=5 loops=1)
         Output: (coordinate <-> '(29.5,-20.6)'::point)
         Order By: (geocoordinate.coordinate <-> '(29.5,-20.6)'::point)
         Heap Fetches: 0
 Planning Time: 1.923 ms
 Execution Time: 0.802 ms
(8 rows)
```

2.4.7 GIN

- GIN stands for **Generalized Inverted Index**. GIN is designed for handling cases where the items to be indexed are composite values, and the queries to be handled by the index need to search for element values that appear within the composite items. For example, the items could be documents, and the queries could be searches for documents containing specific words [19].

- GIN indexes are "inverted indexes" which are appropriate for data values that contain multiple component values, such as arrays. An inverted index contains a separate entry for each component value, and can efficiently handle queries that test for the presence of specific component values.
- Like GiST and SP-GiST, GIN can support many different user-defined indexing strategies, and the particular operators with which a GIN index can be used vary depending on the indexing strategy. As an example, the standard distribution of PostgreSQL includes a GIN operator class for arrays, which supports indexed queries using these operators:

<@	@>	=	&&

- Many other GIN operator classes are available in the `contrib` collection or as separate projects.

2.4.8 BRIN

- BRIN indexes (a shorthand for Block Range INdexes) store summaries about the values stored in consecutive physical block ranges of a table. Thus, they are most effective for columns whose values are well-correlated with the physical order of the table rows. Like GiST, SP-GiST and GIN, BRIN can support many different indexing strategies, and the particular operators with which a BRIN index can be used vary depending on the indexing strategy. For data types that have a linear sort order, the indexed data corresponds to the minimum and maximum values of the values in the column for each block range. This supports indexed queries using these operators:

<	<=	=	>=	>

```
1 CREATE INDEX index_name ON table_name USING brin(column_name);
```

- `SELECT` function without index:

```
test=#
test=# EXPLAIN(ANALYZE,VERBOSE) SELECT * FROM car WHERE price >= '50000';
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on public.car  (cost=0.00..21.50 rows=492 width=30) (actual time=0.029..0.346 rows=492 loops=1)
   Output: id, make, model, year, price
   Filter: (car.price >= '50000'::numeric)
   Rows Removed by Filter: 508
 Planning Time: 0.307 ms
 Execution Time: 0.385 ms
(6 rows)
```

- `SELECT` function with BRIN index:

```
test=# EXPLAIN(ANALYZE,VERBOSE) SELECT * FROM car WHERE price >= '50000';
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Seq Scan on public.car  (cost=0.00..21.50 rows=492 width=30) (actual time=0.015..0.160 rows=492 loops=1)
   Output: id, make, model, year, price
   Filter: (car.price >= '50000'::numeric)
   Rows Removed by Filter: 508
 Planning Time: 0.084 ms
 Execution Time: 0.182 ms
(6 rows)
```

2.4.9 Comparison between B-tree and BRIN index

In this section we try to compare several types of indexes in the same PostgreSQL database with each other.

Loading data sample

Before we dive into the details of B-tree and BRIN indexes, it makes sense to load some sample data. The easiest way to do that is to make use of generate_series. This is the Swiss army knife type of function which is basically used by most PostgreSQL consultants to generate huge tables for testing purposes:

```
1  test=# CREATE TABLE t_demo (id_sorted int8, id_random int8);
2  CREATE TABLE
3  test=# INSERT INTO t_demo
4         SELECT id, hashtext(id::text)
5         FROM generate_series(1, 5000000000) AS id;
6  INSERT 0 5000000000
```

In this case, we created 5 billion rows which lead to a 206 GB table. Here we have two columns: The first column is ascending. It contains numbers ranging from 1 to 5 billion.

```
1  test=# \d+
2  List of relations
3   Schema | Name   | Type  | Owner | Persistence | Access method | Size   | Descr
4  --------+--------+-------+-------+-------------+---------------+--------+------
5   public | t_demo | table | hs    | permanent   | heap          | 206 GB |
6  (1 row)
```

The second column contains a hash value. Take a look:

```
test=# SELECT * FROM t_demo LIMIT 10 OFFSET 50;
 id_sorted | id_random
-----------+-------------
        51 |   342243648
        52 | -1711900017
        53 |   213436769
        54 |  2112769913
        55 | -1318351987
        56 |   -19937162
        57 |  -299365904
        58 | -1228416573
        59 |    93548776
        60 |  1491665190
(10 rows)
```

The point here is that the hash value is pretty much random and should be evenly distributed. It gives us the chance to see how various index types such as B-tree and BRIN behave in these two quite common use cases.

Create index

Let's create a simple B-tree. In a first step we use the PostgreSQL default configuration (the only change I made was to set `max_wal_size` to 100 GB – all other settings are default).

Creating a standard B-tree index will cost us 35 minutes:

```
test=# CREATE INDEX idx_id ON t_demo (id_sorted);
CREATE INDEX
Time: 2109651,552 ms (35:09,652)
```

However, we can do better. If we drop the index again and pump `maintenance_work_mem` to 1 GB, and if we increase the number of worker processes PostgreSQL is allowed to use, we can speed things up:

```
test=# SHOW maintenance_work_mem ;
 maintenance_work_mem
----------------------
 64MB
(1 row)

Time: 0,170 ms
test=# SET maintenance_work_mem TO '1 GB';
SET
Time: 0,181 ms
test=# SET max_parallel_maintenance_workers TO 4;
SET
```

Now let's create the same B-tree index again and see what happens:

```
[hs@hanspc user_test]$ vmstat 2
procs -----------memory---------- ---swap-- -----io---- -system--- ------cpu-----
r b swpd   free   buff cache    si   so bi       bo       in    cs   us sy id wa st
5 0 450048 235408 1800 25689472 0    2  2964     6838     21    41   9  2  88 1  0
5 0 450048 261876 1800 25650596 0    0  1225912  8        19425 8512 39 7  54 0  0
7 0 450048 257548 1800 25660116 0    0  941458   277776   19766 7638 32 7  53 7  0
1 5 450048 232476 1800 25690880 0    0  857176   411110   17791 7603 34 7  50 9  0
5 0 450048 260136 1800 25663332 0    0  756074   485380   16005 7811 28 6  53 13 0
3 4 450048 242340 1800 25679688 0    0  722454   666192   16785 8842 26 10 49 15 0
```

When looking at `vmstat` we see that PostgreSQL starts to read hundreds of MB per second, and starts to spill hundreds of MB/second to disk at a time. Since we cannot sort everything in memory, this is exactly what we expected to happen.

During the first phase we see exactly what happens as revealed by the progress monitoring infrastructure:

```
test=# \x
Expanded display is on.
test=# SELECT * FROM pg_stat_progress_create_index;
-[ RECORD 1 ]------+------------------------------
pid                | 23147
datid              | 24576
datname            | test
relid              | 24583
index_relid        | 0
command            | CREATE INDEX
phase              | building index: scanning table
lockers_total      | 0
lockers_done       | 0
current_locker_pid | 0
blocks_total       | 27027028
blocks_done        | 8577632
tuples_total       | 0
tuples_done        | 0
partitions_total   | 0
partitions_done    | 0
```

PostgreSQL will first scan the table and prepare the data for sorting. In the process table we can see that all desired cores are working at full speed to make this happen:

```
  PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
38830 hs 20 0 594336 219780 12152 R 99,0 0,7 0:06.65 postgres
38831 hs 20 0 594336 219748 12120 R 99,0 0,7 0:06.65 postgres
38833 hs 20 0 594336 219920 12288 R 98,7 0,7 0:06.64 postgres
38832 hs 20 0 594336 219916 12284 R 98,3 0,7 0:06.63 postgres
23147 hs 20 0 514964 276408 149056 R 97,7 0,8 106:48.69 postgres
```

Once this step is done, PostgreSQL can start to sort those partial data sets. Remember, the table is too big to make this happen in memory, so we had to spill to disk:

```
-[ RECORD 1 ]-------+-----------------------------------
pid                 | 23147
datid               | 24576
datname             | test
relid               | 24583
index_relid         | 0
command             | CREATE INDEX
phase               | building index: sorting live tuples
lockers_total       | 0
lockers_done        | 0
current_locker_pid  | 0
blocks_total        | 27027028
blocks_done         | 27027028
tuples_total        | 0
tuples_done         | 0
partitions_total    | 0
partitions_done     | 0
```

After this stage PostgreSQL will add data to the tree and write the index to disk:

```
-[ RECORD 1 ]-------+--------------------------------------
pid                 | 23147
datid               | 24576
datname             | test
relid               | 24583
index_relid         | 0
command             | CREATE INDEX
phase               | building index: loading tuples in tree
lockers_total       | 0
lockers_done        | 0
current_locker_pid  | 0
blocks_total        | 0
blocks_done         | 0
tuples_total        | 5000000000
tuples_done         | 2005424786
partitions_total    | 0
partitions_done     | 0
```

We managed to almost double the performance of our indexing process – which is a really great achievement:

```
test=# CREATE INDEX idx_id ON t_demo (id_sorted);
CREATE INDEX
Time: 1212601,300 ms (20:12,601)
```

What is noteworthy here is the size of the index we have just created:

```
test=# \di+
List of relations
 Schema | Name   | Type  | Owner | Table  | Persistence | Access method | Size
--------+--------+-------+-------+--------+-------------+---------------+--------
 public | idx_id | index | hs    | t_demo | permanent   | btree         | 105 GB
(1 row)
```

105 GB is pretty massive. However, it basically confirms the following rule of thumb: Usually an index needs around 25 bytes per index entry – assuming we do not have any duplicate entries.

Create a second index:

```
test=# CREATE INDEX idx_random ON t_demo (id_random);
CREATE INDEX
Time: 1731088,437 ms (28:51,088)
```

In this case, we have indexed the randomized column. As our input stream is not sorted anymore it takes a bit longer to create this index.

Running queries using B-tree

Run a query and see what the B-tree index has done so far:

```
test=# SELECT count(*)
FROM t_demo
WHERE id_sorted BETWEEN 10000000 AND 20000000;
   count
----------
 10000001
(1 row)

Time: 358,804 ms
```

What we see here is stunning performance. We managed to extract 10 million rows in just 358 milliseconds. Let's look deep into the execution plans and see what happens:

```
test=# explain (analyze, buffers) SELECT count(*)
       FROM  t_demo
       WHERE id_sorted BETWEEN 10000000 AND 20000000;
                                QUERY PLAN
-------------------------------------------------------------------------
 Finalize Aggregate (cost=296509.46..296509.47 rows=1 width=8)
                    (actual time=496.450..500.609 rows=1 loops=1)
                   Buffers: shared hit=11 read=27325
    ->  Gather   (cost=296509.24..296509.45 rows=2 width=8)
                 (actual time=496.379..500.603 rows=3 loops=1)
          Workers Planned: 2
          Workers Launched: 2
          Buffers: shared hit=11 read=27325
          ->  Partial Aggregate  (cost=295509.24..295509.25 rows=1 width=8)
                                 (actual time=492.621..492.622 rows=1 loops=3)
                Buffers: shared hit=11 read=27325
                ->  Parallel Index Only Scan using idx_id on t_demo
                          (cost=0.58..284966.32 rows=4217169 width=0)
                          (actual time=0.074..340.666 rows=3333334 loops=3)
                      Index Cond: ((id_sorted >= 10000000)
                            AND (id_sorted <= 20000000))
                      Heap Fetches: 0
                      Buffers: shared hit=11 read=27325
 Planning:
   Buffers: shared hit=7 read=3
 Planning Time: 0.233 ms
 Execution Time: 500.662 ms
(16 rows)
```

The above is a parallel-index-only scan. Actually this is good news because it means that it is enough to scan the index. There is no need to fetch data from the underlying table. All we need to ensure is the existence of the row (which is in this case done through PostgreSQL hint bits).

Now we run the same query, but this time we use the random column. Note that we reduced the range a bit to ensure that we also get 10 million rows. The size of the result set is therefore roughly identical:

```
test=# explain SELECT count(*)
       FROM  t_demo
       WHERE id_random BETWEEN 10000000 AND 19000000;
                                QUERY PLAN
-------------------------------------------------------------------------
 Aggregate  (cost=22871994.27..22871994.28 rows=1 width=8)
   ->  Index Only Scan using idx_random on t_demo
            (cost=0.58..22846435.29 rows=10223592 width=0)
         Index Cond: ((id_random >= 10000000)
           AND (id_random <= 19000000))
(3 rows)
```

The parallel query using more than one core is gone. However, it is even worse to see what happens during the query:

```
[hs@hanspc user_test]$ vmstat 2
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  1 1377280 228168   1032 27286216    0    0 120584   314 17933 30276  2  5 86
 1  0 1377280 231608   1032 27286224    0    0 121102  1069 17839 30854  2  3 88
 1  1 1377280 251400   1032 27270360    0    0 121168   869 17669 30177  1  4 88
 0  1 1377280 263096   1032 27254396    0    0 121674   514 17780 30695  2  2 88
 0  1 1377280 236788   1032 27275712    0    0 121630   260 18403 31970  2  2 88
```

We can see that 120 MB/sec sustained I/O which translates to horrible runtime. Note that the amount of data is the same – all that has changes is the distribution of data on disk:

```
test=# SELECT count(*)
       FROM t_demo
       WHERE id_random BETWEEN 10000000 AND 19000000;
  count
----------
 10481155
(1 row)
Time: 322747,613 ms (05:22,748)
```

5 minutes. As we can see here our runtime has exploded. We can figure out why:

```
test=# explain (analyze, buffers) SELECT count(*)
       FROM  t_demo
       WHERE id_random BETWEEN 10000000 AND 19000000;
                                QUERY PLAN
-------------------------------------------------------------------------
 Aggregate  (cost=22871994.27..22871994.28 rows=1 width=8)
            (actual time=292381.841..292381.843 rows=1 loops=1)
            Buffers: shared hit=6611938 read=3884860
   ->  Index Only Scan using idx_random on t_demo
        (cost=0.58..22846435.29 rows=10223592 width=0)
        (actual time=0.515..291336.908 rows=10481155 loops=1)
         Index Cond: ((id_random >= 10000000)
                 AND (id_random <= 19000000))
         Heap Fetches: 3866504
         Buffers: shared hit=6611938 read=3884860
 Planning Time: 1.080 ms
 Execution Time: 292381.934 ms
(8 rows)

Time: 292384,903 ms (04:52,385)
```

Data is spread all over the place. Therefore we need millions of 8 kB pages to handle the query. In those blocks we only need a subset of data. That causes substantial runtime.

BRIN indexes

BRIN indexes are often seen as a savior for warehousing applications. However, you should not be so sure about that.

First, we create an index:

```
test=# CREATE INDEX idx_brin ON t_demo USING brin (id_sorted);
CREATE INDEX
Time: 710549,637 ms (11:50,550)
```

The index is made really quickly. What is also extremely convincing is the size of the BRIN index:

```
test=# SELECT pg_size_pretty(pg_relation_size('idx_brin'));
 pg_size_pretty
----------------
 7064 kB
(1 row)
```

It is only 7 MB which saves us a stunning 10 GB on disk. What we can also see during index creation is that the process looks slightly different:

```
test=# SELECT * FROM pg_stat_progress_create_index;
-[ RECORD 1 ]------+---------------
pid                | 23147
datid              | 24576
datname            | test
relid              | 24583
index_relid        | 0
command            | CREATE INDEX
phase              | building index
lockers_total      | 0
lockers_done       | 0
current_locker_pid | 0
blocks_total       | 27027028
blocks_done        | 16898015
tuples_total       | 0
tuples_done        | 0
partitions_total   | 0
partitions_done    | 0
```

BRIN is not as extensive a sorting exercise as normal B-tree index creation. This is important because it translates to less temporary I/O and faster index creation.

Compare queries

```
test=# explain (analyze, buffers) SELECT count(*)
       FROM  t_demo
       WHERE id_sorted BETWEEN 10000000 AND 20000000;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Finalize Aggregate (cost=49930967.53..49930967.54 rows=1 width=8)
                    (actual time=664.605..671.519 rows=1 loops=1)
                    Buffers: shared hit=311 read=55025
    ->  Gather  (cost=49930967.32..49930967.53 rows=2 width=8)
                (actual time=664.387..671.502 rows=3 loops=1)
          Workers Planned: 2
          Workers Launched: 2
          Buffers: shared hit=311 read=55025
          ->  Partial Aggregate (cost=49929967.32..49929967.33 rows=1 width=8)
                                (actual time=653.953..653.955 rows=1 loops=3)
                Buffers: shared hit=311 read=55025
                ->  Parallel Bitmap Heap Scan on t_demo
                      (cost=5390.21..49920339.39 rows=3851169 width=0)
                      (actual time=56.944..489.563 rows=3333334 loops=3)
                    Recheck Cond: ((id_sorted >= 10000000)
                                   AND (id_sorted <= 20000000))
                    Rows Removed by Index Recheck: 5546
                    Heap Blocks: lossy=19046
                    Buffers: shared hit=311 read=55025
                    ->  Bitmap Index Scan on idx_brin
                              (cost=0.00..3079.51 rows=9258865 width=0)
                              (actual time=65.293..65.294 rows=541440 loops=1)
                          Index Cond: ((id_sorted >= 10000000)
                                       AND (id_sorted <= 20000000))
                          Buffers: shared hit=311 read=881
  Planning:
    Buffers: shared hit=1
  Planning Time: 0.210 ms
  Execution Time: 671.623 ms
 (20 rows)
```

The B-tree managed to run the query with just 27,000 blocks – now we need around 55,000 – which translates to more runtime.

While a BRIN index is usually a good idea for such use cases, it is not always the magic bullet. It does need significantly less space and it can be built faster. But is does not magically fix all cases under all circumstances.

Conclusion

Please keep in mind that BRIN is not bad – it is usually a speedup compared to B-tree indexes. We should compare various setups and decide what is best.

2.5 Query Processing

We divide this section into 2 parts, the first part is the basics of Query Processing in PostgreSQL and the second part is detailed through the processing example.

2.5.1 The Query Tree

In PostgreSQL, the rule system is located between the parser and the planner. It takes the output of the parser, one query tree, and the user-defined rewrite rules, which are also query trees with some extra information, and creates zero or more query trees as result. So its input and output are always things the parser itself could have produced and thus, anything it sees is basically representable as an SQL statement. A query tree is an internal representation of an SQL statement where the single parts that it is built from are stored separately. These query trees can be shown in the server log if you set the configuration parameters `debug_print_parse`, `debug_print_rewritten`, or `debug_print_plan`. The rule actions are also stored as query trees, in the system catalog `pg_rewrite`. They are not formatted like the log output, but they contain exactly the same information. When reading the SQL representations of the query trees in this part it is necessary to be able to identify the parts the statement is broken into when it is in the query tree structure. The parts of a query tree are

- `the command type`
  This is a simple value telling which command `(SELECT, INSERT, UPDATE, DELETE)` produced the query tree.

- `the range table`
  The range table is a list of relations that are used in the query. In a `SELECT` statement these are the relations given after the `FROM` key word.

- `the result relation`
  This is an index into the range table that identifies the relation where the results of the query go.
  `SELECT` queries do not have a result relation. (The special case of `SELECT INTO` is mostly identical to `CREATE TABLE` followed by `INSERT ... SELECT`, and is not discussed separately here.) For `INSERT`, `UPDATE`, and `DELETE commands`, the result relation is the table (or view!) where the changes are to take effect.

- `the target list`
  The target list is a list of expressions that define the result of the query. In the case of a `SELECT`, these expressions are the ones that build the final output of the query. They correspond to the expressions between the key words `SELECT` and `FROM`. (`*` is just an abbreviation for all the column names of a relation. It is expanded by the parser into the individual columns, so the rule system never sees it.)

  - `DELETE` commands do not need a normal target list because they do not produce any result. Instead, the planner adds a special CTID (a hidden and unique record for each table in PostgreSQL) entry to the empty target list, to allow the executor to find the row to be deleted. (CTID is added when the result relation is an ordinary table.
  - For `INSERT` commands, the target list describes the new rows that should go into the result relation. It consists of the expressions in the `VALUES` clause or the ones from the `SELECT` clause in `INSERT ... SELECT`. The first step of the rewrite process adds target list entries for any columns that were not assigned to by the original command but have defaults. Any remaining columns (with neither a given value nor a default) will be filled in by the planner with a constant null expression.
  - For `UPDATE` commands, the target list describes the new rows that should replace the old ones. In the rule system, it contains just the expressions from the `SET column = expression` part of the command. The planner will handle missing columns by

inserting expressions that copy the values from the old row into the new one. Just as for DELETE, a CTID or whole-row variable is added so that the executor can identify the old row to be updated.

– Every entry in the target list contains an expression that can be a constant value, a variable pointing to a column of one of the relations in the range table, a parameter, or an expression tree made of function calls, constants, variables, operators, etc.

- the qualification
  The query's qualification is an expression much like one of those contained in the target list entries. The result value of this expression is a Boolean that tells whether the operation (INSERT, UPDATE, DELETE, or SELECT) for the final result row should be executed or not. It corresponds to the WHERE clause of an SQL statement.

- the join tree
  The query's join tree shows the structure of the FROM clause. For a simple query like SELECT ... FROM a, b, c, the join tree is just a list of the FROM items, because we are allowed to join them in any order. But when JOIN expressions, particularly outer joins, are used, we have to join in the order shown by the joins. In that case, the join tree shows the structure of the JOIN expressions. The restrictions associated with particular JOIN clauses (from ON or USING expressions) are stored as qualification expressions attached to those join-tree nodes. It turns out to be convenient to store the top-level WHERE expression as a qualification attached to the top-level join-tree item, too. So really the join tree represents both the FROM and WHERE clauses of a SELECT.

- the others
  The other parts of the query tree like the ORDER BY clause are not of interest here. The rule system substitutes some entries there while applying rules, but that does not have much to do with the fundamentals of the rule system.

2.5.2 The Planner

The task of the planner/optimizer is to create an optimal execution plan. A given SQL query (and hence, a query tree) can be actually executed in a wide variety of different ways, each of which will produce the same set of results. If it is computationally feasible, the query optimizer will examine each of these possible execution plans, ultimately selecting the execution plan that is expected to run the fastest. The planner's search procedure actually works with data structures called **paths**, which are simply cut-down representations of plans containing only as much information as the planner needs to make its decisions. After the cheapest path is determined, a full-fledged plan tree is built to pass to the executor. This represents the desired execution plan in sufficient detail for the executor to run it. In the rest of this section we ignore the distinction between paths and plans. The planner/optimizer starts by generating plans for scanning each individual relation (table) used in the query. The possible plans are determined by the available indexes on each relation. There is always the possibility of performing a sequential scan on a relation, so a sequential scan plan is always created. If the query requires joining two or more relations, plans for joining relations are considered after all feasible plans have been found for scanning single relations. The three available join strategies are:

- nested loop join
  The right relation is scanned once for every row found in the left relation. This strategy is easy to implement but can be very time consuming. (However, if the right relation can be

scanned with an index scan, this can be a good strategy. It is possible to use values from the current row of the left relation as keys for the index scan of the right.)

- `merge join`
  Each relation is sorted on the join attributes before the join starts. Then the two relations are scanned in parallel, and matching rows are combined to form join rows. This kind of join is attractive because each relation has to be scanned only once. The required sorting might be achieved either by an explicit sort step, or by scanning the relation in the proper order using an index on the join key.

- `hash join`
  When the query involves more than two relations, the final result must be built up by a tree of join steps, each with two inputs. The planner examines different possible join sequences to find the cheapest one.

The finished plan tree consists of sequential or index scans of the base relations, plus nested-loop, merge, or hash join nodes as needed, plus any auxiliary steps needed, such as sort nodes or aggregate-function calculation nodes. Most of these plan node types have the additional ability to do selection (discarding rows that do not meet a specified Boolean condition) and projection (computation of a derived column set based on given column values, that is, evaluation of scalar expressions where needed). One of the responsibilities of the planner is to attach selection conditions from the `WHERE` clause and computation of required output expressions to the most appropriate nodes of the plan tree.

2.6 Query Execution Stages

The fundamental purpose of the PostgreSQL client-server protocol is twofold: it sends SQL queries (from client or frontend) to the server (backend), and it receives the entire execution result in response. The query received by the server for execution goes through several stages [23] that will be discussed in the following sections. Note that in this section, we only demonstrate the **simple query protocol** with default database from PostgreSQL:

```
1 CREATE TABLE pg_tables (schemaname VARCHAR(128), tablename VARCHAR(128), tableowner
  VARCHAR(128));
```

The backend process basically handles all queries issued by the connected client. This backend consists of five subsystems, we group them into 3 parts:

- **Parsing**
  - Parser
    The parser generates a parse tree from an SQL statement in plain text.
  - Analyzer/Analyser
    The analyzer/analyser carries out a semantic analysis of a parse tree and generates a query tree.
  - Rewriter/Transformation
    The rewriter transforms a query tree using the rules stored in the rule system if such rules exist.

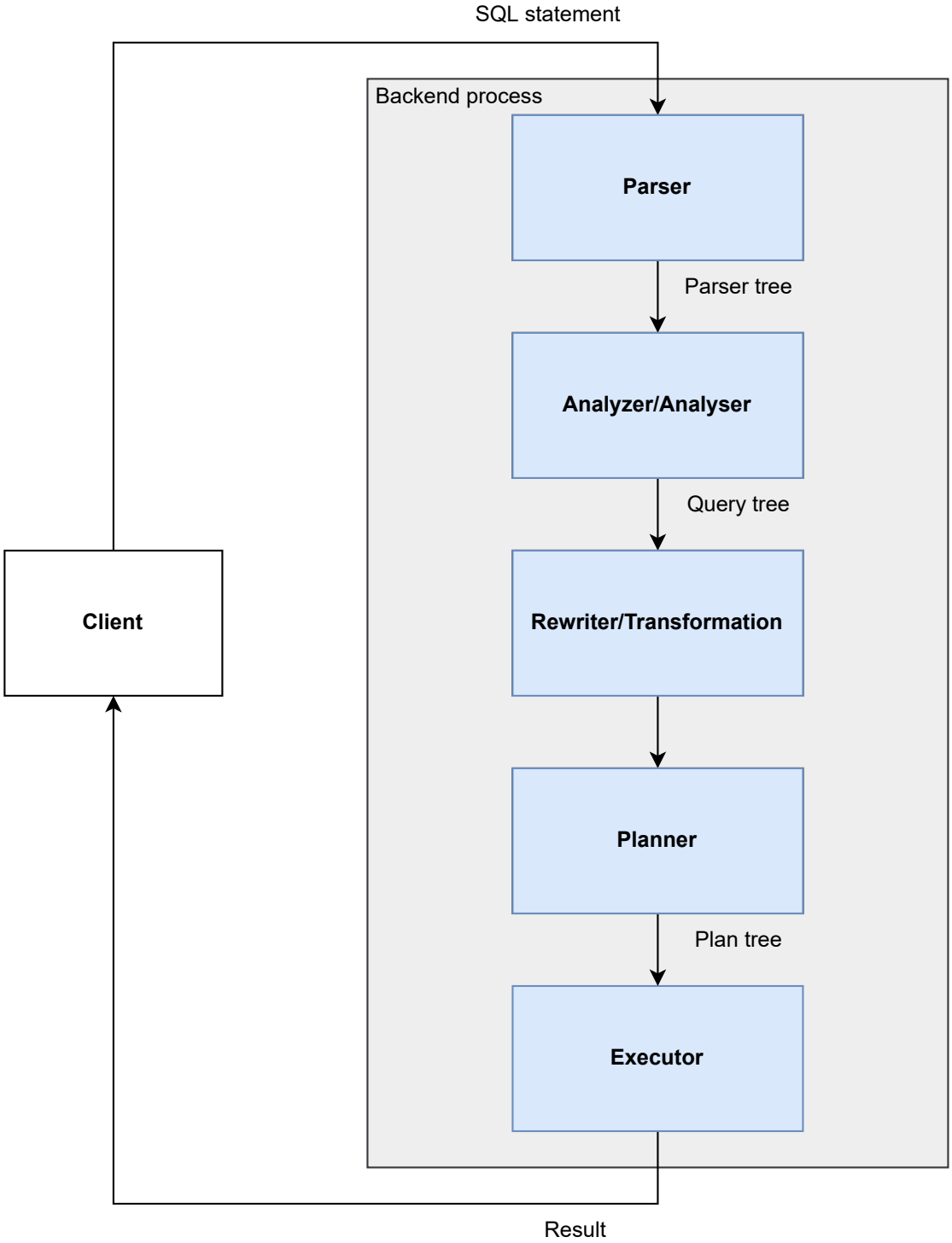
- **Planning**

  - Planner
    The planner generates the plan tree that can most effectively be executed from the query tree.

- **Execution**

  - Executor
    The executor executes the query via accessing the tables and indexes in the order that was created by the plan tree.

As shown below:


SQL statement
Backend process
Parser
Parser tree
Analyzer/Analyser
Query tree
Rewriter/Transformation
Planner
Plan tree
Executor
Client
Result

2.6.1 Parsing

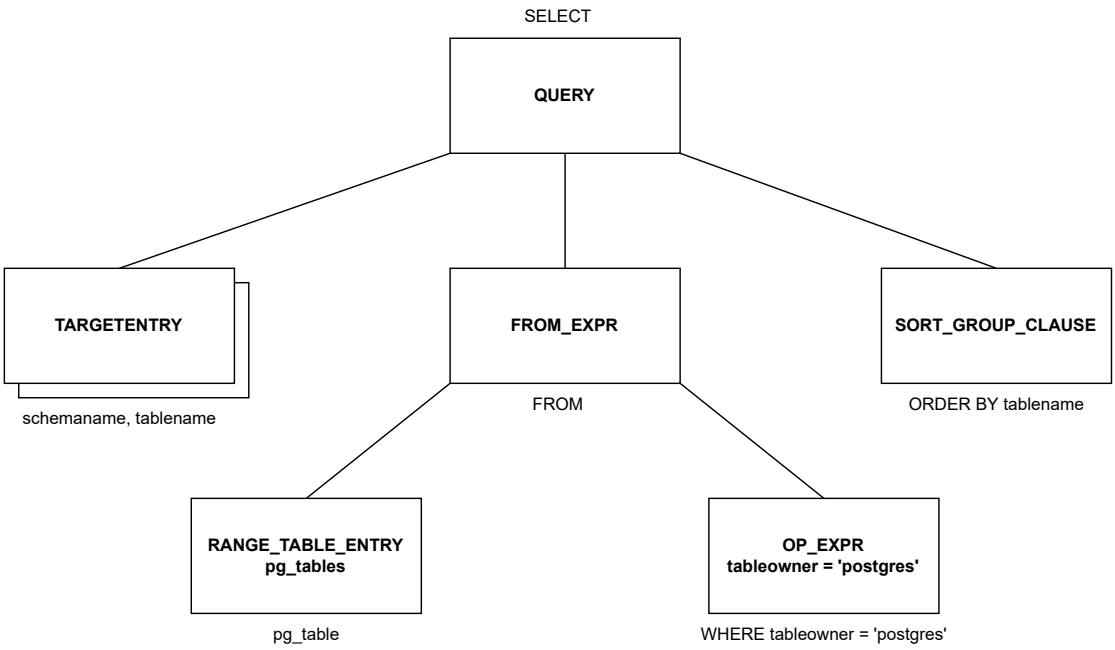
First, the query text is parsed, so that the server understands exactly what needs to be done.

2.6.1.a Lexer and Parser

The lexer is responsible for recognizing lexemes in the query string (such as SQL keywords, string and numeric literals, etc.), and the parser makes sure that the resulting set of lexemes is grammatically valid. The parser and lexer are implemented using the standard tools Bison and Flex.
The parsed query is represented as an abstract syntax tree. Example:

```
1 SELECT schemaname, tablename
2 FROM pg_tables
3 WHERE tableowner = 'postgres'
4 ORDER BY tablename;
```

Here, a tree will be built in backend memory. The figure below shows the tree in a highly simplified form. The nodes of the tree are labeled with the corresponding parts of the query. The parser generates a parse tree that can be read by subsequent subsystems from an SQL statement in plain text.



The name “range table” in the PostgreSQL source code refers to tables, subqueries, results of joins—in other words, any record sets that SQL statements operate on. The parser does not check the semantics of an input query. For example, even if the query contains a table name that does not exist, the parser does not return an error. Semantic checks are done by the analyzer/analyser.

2.6.1.b Semantic Analyzer

The analyzer/analyser runs a semantic analysis of a parse tree generated by the parser and generates a query tree.

The semantic analyzer determines whether there are tables and other objects in the database that the query refers to by name, and whether the user has the right to access these objects. All the information required for semantic analysis is stored in the system catalog.

The semantic analyzer receives the parse tree from the parser and rebuilds it, supplementing it with references to specific database objects, data type information, etc.

If the parameter `debug_print_parse` is on, the full tree will be displayed in the server message log, although there is little practical sense in this.

The root of a query tree is the Query structure defined in `parsenodes.h`; this structure contains metadata of its corresponding query such as the type of this command (`SELECT`, `INSERT` or others) and several leaves; each leaf forms a list or a tree and holds data of the individual particular clause.

2.6.1.c Transformation

Next, the query can be transformed (**rewritten**).

The rewriter is the system that realizes the rule system, and transforms a query tree according to the rules stored in the `pg_rules` system catalog if necessary. The rule system is an interesting system in itself, however, the descriptions of the rule system and the rewriter have been omitted to prevent this report from becoming too long.

Transformations are used by the system core for several purposes. One of them is to replace the name of a view in the parse tree with a subtree corresponding to the query of this view.

`pg_tables` from the example above is a view, and after transformation the parse tree will take the following form:

This parse tree corresponds to the following query (although all manipulations are performed only on the tree, not on the query text):

```sql
SELECT schemaname, tablename
FROM (
    -- pg_tables
    SELECT n.nspname AS schemaname,
      c.relname AS tablename,
      pg_get_userbyid(c.relowner) AS tableowner,
      ...
    FROM pg_class c
      LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
      LEFT JOIN pg_tablespace t ON t.oid = c.reltablespace
    WHERE c.relkind = ANY (ARRAY['r'::char, 'p'::char])
)
```

```
13 WHERE tableowner = 'postgres'
14 ORDER BY tablename;
```

- The parse tree reflects the syntactic structure of the query, but not the order in which the operations will be performed.
- Row-level security is implemented at the transformation stage.
- Another example of the use of transformations by the system core is the implementation of `SEARCH` and `CYCLE` clauses for recursive queries in version 14.
- The rule system was intended as one of the primary features of PostgreSQL. The rules were supported from the project's foundation and were repeatedly redesigned during early development. This is a powerful mechanism but difficult to understand and debug. There was even a proposal to remove the rules from PostgreSQL entirely, but it did not find general support. In most cases, it is safer and more convenient to use triggers instead of rules.
- If the parameter `debug_print_rewritten` is on, the complete transformed parse tree will be displayed in the server message log.

2.6.2 Planning

SQL is a declarative language: a query specifies what to retrieve, but not how to retrieve it.

Any query can be executed in a number of ways. Each operation in the parse tree has multiple execution options. For example, user can retrieve specific records from a table by reading the whole table and discarding rows user do not need, or user can use indexes to find the rows that match your query. Data sets are always joined in pairs. Variations in the order of joins result in a multitude of execution options. Then there are various ways to join two sets of rows together. For example, user could go through the rows in the first set one by one and look for matching rows in the other set, or user could sort both sets first, and then merge them together. Different approaches perform better in some cases and worse in others.

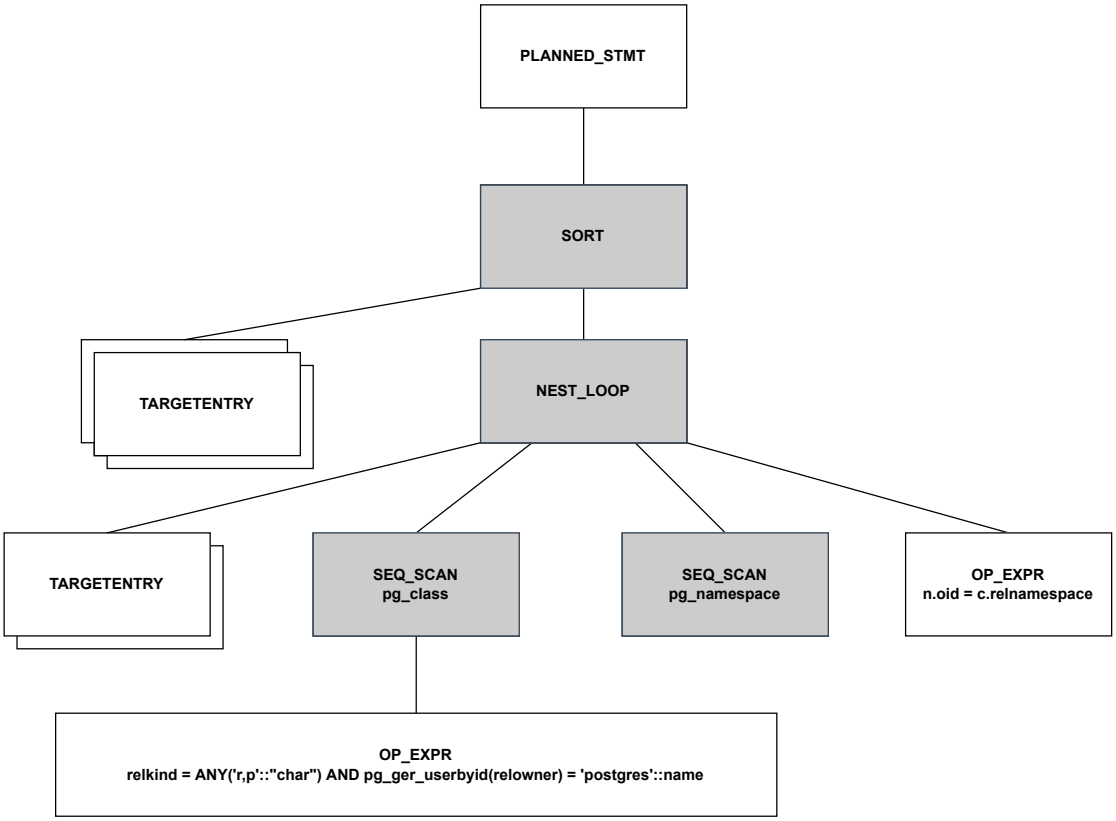
The optimal plan may execute faster than a non-optimal one by several orders of magnitude. This is why the planner, which optimizes the parsed query, is one of the most complex elements of the system.

The planner receives a query tree from the rewriter and generates a (query) plan tree that can be processed by the executor most effectively.

The planner in PostgreSQL is based on pure cost-based optimization; **it does not support rule-based optimization and hints**. This planner is the most complex subsystem in RDBMS (Relational Database Management System).

2.6.2.a Plan Tree

The execution plan can also be presented as a tree, but with its nodes as physical rather than logical operations on data.




```
1 EXPLAIN
2 SELECT schemaname, tablename
3 FROM pg_tables
4 WHERE tableowner = 'postgres'
5 ORDER BY tablename;
```

```
1                                   QUERY PLAN
2 ------------------------------------------------------------------------------
3  Sort  (cost=21.03..21.04 rows=1 width=128)
4    Sort Key: c.relname
5    ->  Nested Loop Left Join  (cost=0.00..21.02 rows=1 width=128)
6          Join Filter: (n.oid = c.relnamespace)
7          ->  Seq Scan on pg_class c  (cost=0.00..19.93 rows=1 width=72)
8                Filter: ((relkind = ANY ('{r,p}'::"char"[])) AND
                 (pg_get_userbyid(relowner) = 'postgres'::name))
9          ->  Seq Scan on pg_namespace n  (cost=0.00..1.04 rows=4 width=68)
10 (7 rows)
```

The figure shows the main nodes of the tree. The same nodes are marked with arrows in the `EXPLAIN` output.

The `SEQ_SCAN` node represents the table read operation, while the `NEST_NODE` node represents the join operation. There are two interesting points to take note of here:

- One of the initial tables is gone from the plan tree because the planner figured out that it is not required to process the query and removed it.
- There is an estimated number of rows to process and the cost of processing next to each node.

2.6.2.b Plan Search

To find the optimal plan, PostgreSQL utilizes the cost-based query optimizer. The optimizer goes through various available execution plans and estimates the required amounts of resources, such as I/O operations and CPU cycles. This calculated estimate, converted into arbitrary units, is known as the plan cost. The plan with the lowest resulting cost is selected for execution.

The trouble is, the number of possible plans grows exponentially as the number of joins increases, and sifting through all the plans one by one is impossible even for relatively simple queries. Therefore, dynamic programming and heuristics are used to limit the scope of search. This allows to precisely solve the problem for a greater number of tables in a query within reasonable time, but the selected plan is not guaranteed to be truly optimal because the planner utilizes simplified mathematical models and may use imprecise initial data.

2.6.2.c Ordering Joins

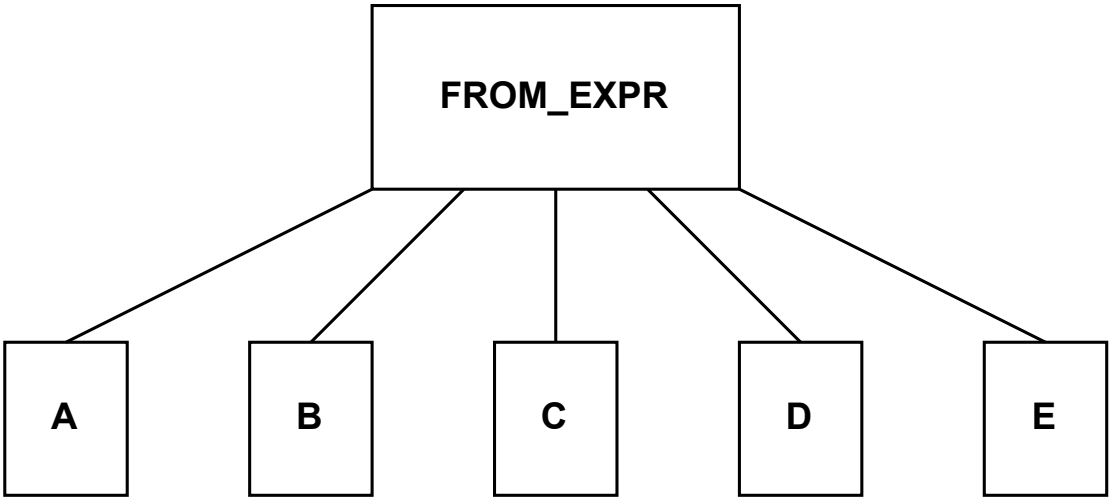
A query can be structured in specific ways to significantly reduce the search scope (at a risk of missing the opportunity to find the optimal plan):

- Common table expressions are usually optimized separately from the main query. Since version 12, this can be forced with the `MATERIALIZE` clause.
- Queries from non-SQL functions are optimized separately from the main query. (SQL functions can be inlined into the main query in some cases.)
- The `join_collapse_limit` parameter together with explicit `JOIN` clauses, as well as the `from_collapse_limit` parameter together with sub-queries may define the order of some joins, depending on the query syntax.

The last one may need an explanation. The query below calls several tables within a FROM clause with no explicit joins:

```
1 SELECT ...
2 FROM a, b, c, d, e
3 WHERE ...
```

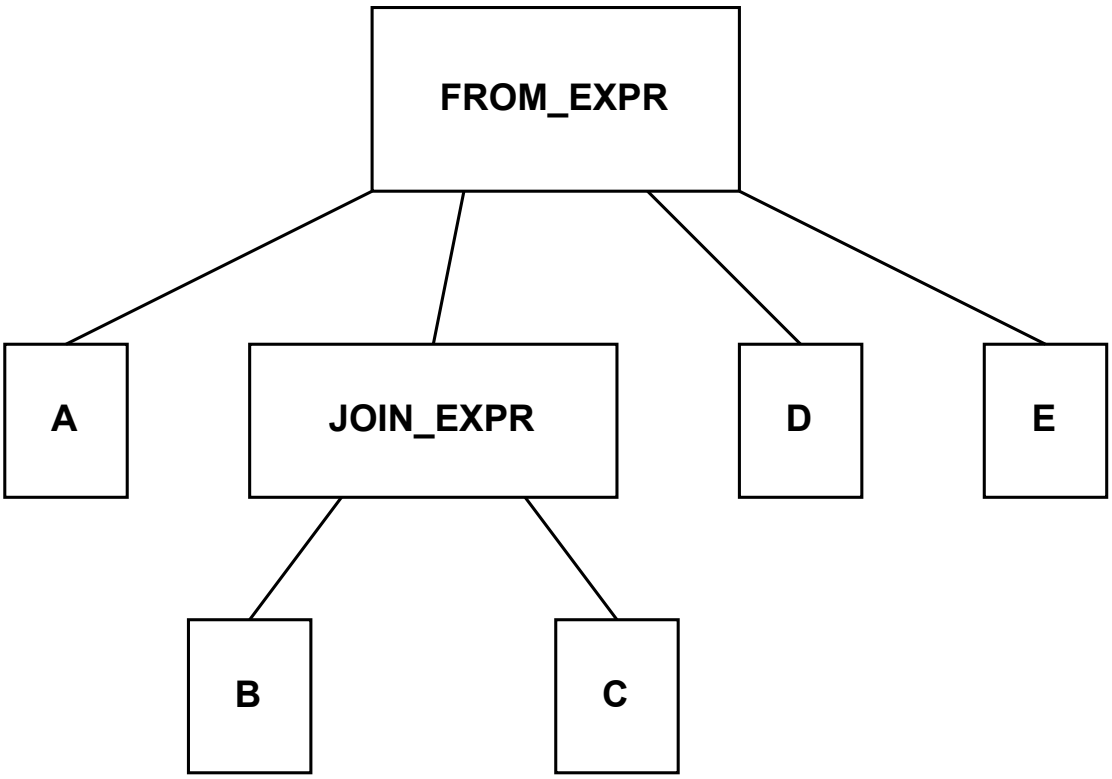
This is the parse tree for this query:



In this query, the planner will consider all possible join orders. In the next example, some joins are explicitly defined by the `JOIN` clause:

```
SELECT ...
FROM a, b JOIN c ON ..., d, e
WHERE ...
```

The parse tree reflects this:



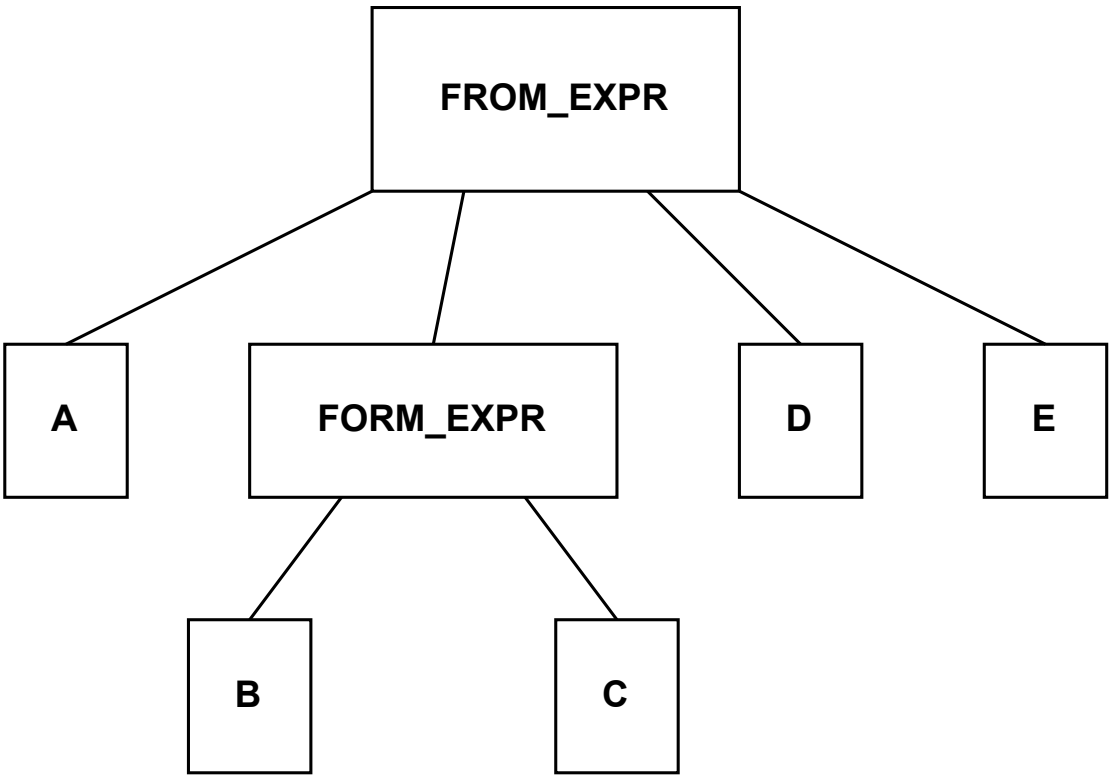
The planner collapses the join tree, effectively transforming it into the tree from the previous example. The algorithm recursively traverses the tree and replaces each `JOIN_EXPR` node with a flat list of its components.
This "flattening out" will only occur, however, if the resulting flat list will contain no more than `join_collapse_limit` elements (8 by default). In the example above, if `join_collapse_limit` is set to 5 or less, the `JOIN_EXPR` node will not be collapsed. For the planner this means two things:

- Table `B` must be joined to table `C` (or vice versa, the join order in a pair is not restricted).
- Tables `A`, `D`, `E`, and the join of `B` to `C` may be joined in any order.

If `join_collapse_limit` is set to 1, any explicit `JOIN` order will be preserved. Note that the operation `FULL OUTER JOIN` is never collapsed regardless of `join_collapse_limit`.
The parameter `from_collapse_limit` (also 8 by default) limits the flattening of sub-queries in a similar manner. Sub-queries do not appear to have much in common with joins, but when it comes down to the parse tree level, the similarity is apparent. Example:

```
1 SELECT ...
2 FROM a, b JOIN c ON ..., d, e
3 WHERE ...
```

The parse tree reflects this:



The only difference here is that the `JOIN_EXPR` node is replaced with `FROM_EXPR` (hence the parameter name `FROM`).

2.6.2.d Genetic Search

Whenever the resulting flattened tree ends up with too many same-level nodes (tables or join results), planning time may skyrocket because each node requires separate optimization. If the parameter geqo is on (it is by default), PostgreSQL will switch to genetic search whenever the number of same-level nodes reaches `geqo_threshold` (12 by default). Genetic search is much faster than the dynamic programming approach, but it does not guarantee that the best possible plan will be found.

2.6.2.e Selection for the Best Plan

The definition of the best plan varies depending on the intended use. When a complete output is required (for example, to generate a report), the plan must optimize the retrieval of all rows that match the query. On the other hand, if you only want the first several matching rows (to display on the screen, for example), the optimal plan might be completely different. PostgreSQL addresses this by calculating two cost components. They are displayed in the query plan output after the word "cost":

```
1 (cost=21.03..21.04 rows=1 width=128)
```

- The first component, or startup cost, is the cost to prepare for the execution of the node.

- The second component, or total cost, represents the total node execution cost.

When selecting a plan, the planner first checks if a cursor is in use (a cursor can be set up with the `DECLARE` command or explicitly declared in PL/pgSQL). If not, the planner assumes that the full output is required and selects the plan with the least total cost.
Otherwise, if a cursor is used, the planner selects a plan that optimally retrieves the number of rows equal to `cursor_tuple_fraction` (0.1 by default) of the total number of matching rows. Or, more specifically, a plan with the lowest cost based on this equation:

```
startup_cost + cursor_tuple_fraction × (total_cost - startup_cost)
```

2.6.2.f Cost Calculation Process

To estimate a plan cost, each of its nodes has to be individually estimated. A node cost depends on the node type (reading from a table costs much less than sorting the table) and the amount of data processed (in general, the more data, the higher the cost). While the node type is known right away, to assess the amount of data we first need to estimate the node's cardinality (the amount of input rows) and selectivity (the fraction of rows left over for output). To do that, we need data statistics: table sizes, data distribution across columns.

Therefore, optimization depends on accurate statistics, which are gathered and kept up-to-date by the autoanalyze process.

If the cardinality of each plan node is estimated accurately, the total cost calculated will usually match the actual cost. Common planner deviations are usually the result of incorrect estimation of cardinality and selectivity. These errors are caused by inaccurate, outdated or unavailable statistics data, and, to a lesser extent, inherently imperfect models the planner is based on.

2.6.2.g Cardinality Estimation

Cardinality estimation is performed recursively. Node cardinality is calculated using two values:

- Cardinality of the node's child nodes, or the number of input rows.
- Selectivity of the node, or the fraction of output rows to the input rows.

Cardinality is the product of these two values.

Selectivity is a number between 0 and 1. Selectivity values closer to zero are called high selectivity, and values closer to one are called low selectivity. This is because high selectivity eliminates a higher fraction of rows, and lower selectivity values bring the threshold down, so fewer rows are discarded.

Leaf nodes with data access methods are processed first. This is where statistics such as table sizes come in.

Selectivity of conditions applied to a table depends on the condition types. In its simplest form selectivity can be a constant value, but the planner tries to use all available information to produce the most accurate estimate. Selectivity estimations for the simplest conditions serve as the basis, and complex conditions built with Boolean operations can be further calculated using

the following straightforward formulas:

$$sel_{x\ and\ y} = sel_x sel_y$$
$$sel_{x\ or\ y} = 1 - (1 - sel_x)(1 - sel_y) = sel_x + sel_y - sel_x sel_y$$

In these formulas, x and y are considered independent. If they correlate, the formulas are still used, but the estimate will be less accurate.

For a cardinality estimate of joins, two values are calculated: the cardinality of the Cartesian product (the product of cardinalities of two data sets) and the selectivity of the join conditions, which in turn depends on the condition types.

Cardinality of other node types, such as sorting or aggregation nodes, is calculated similarly.

Note that a cardinality calculation mistake in a lower node will propagate upward, resulting in inaccurate cost estimation and, ultimately, a sub-optimal plan. This is made worse by the fact that the planner only has statistical data on tables, not on join results.

2.6.2.h Cost Estimation

In PostgreSQL, there are three kinds of costs: start-up, run and total. The total cost is the sum of start-up and run costs; thus, only the start-up and run costs are independently estimated.

- The start-up cost is the cost expended before the first tuple is fetched. For example, the start-up cost of the index scan node is the cost to read index pages to access the first tuple in the target table.
- The run cost is the cost to fetch all tuples.
- The total cost is the sum of the costs of both start-up and run costs.

Cost estimation process is also recursive. The cost of a sub-tree comprises the costs of its child nodes plus the cost of the parent node.

Node cost calculation is based on a mathematical model of the operation it performs. The cardinality, which has been already calculated, serves as the input. The process calculates both startup cost and total cost.

Some operations do not require any preparation and can start executing immediately. For these operations, the startup cost will be zero.

Other operations may have prerequisites. For example, a sorting node will usually require all of the data from its child node to begin the operation. These nodes have a non-zero startup cost. This cost has to be paid, even if the next node (or the client) only needs a single row of the output.

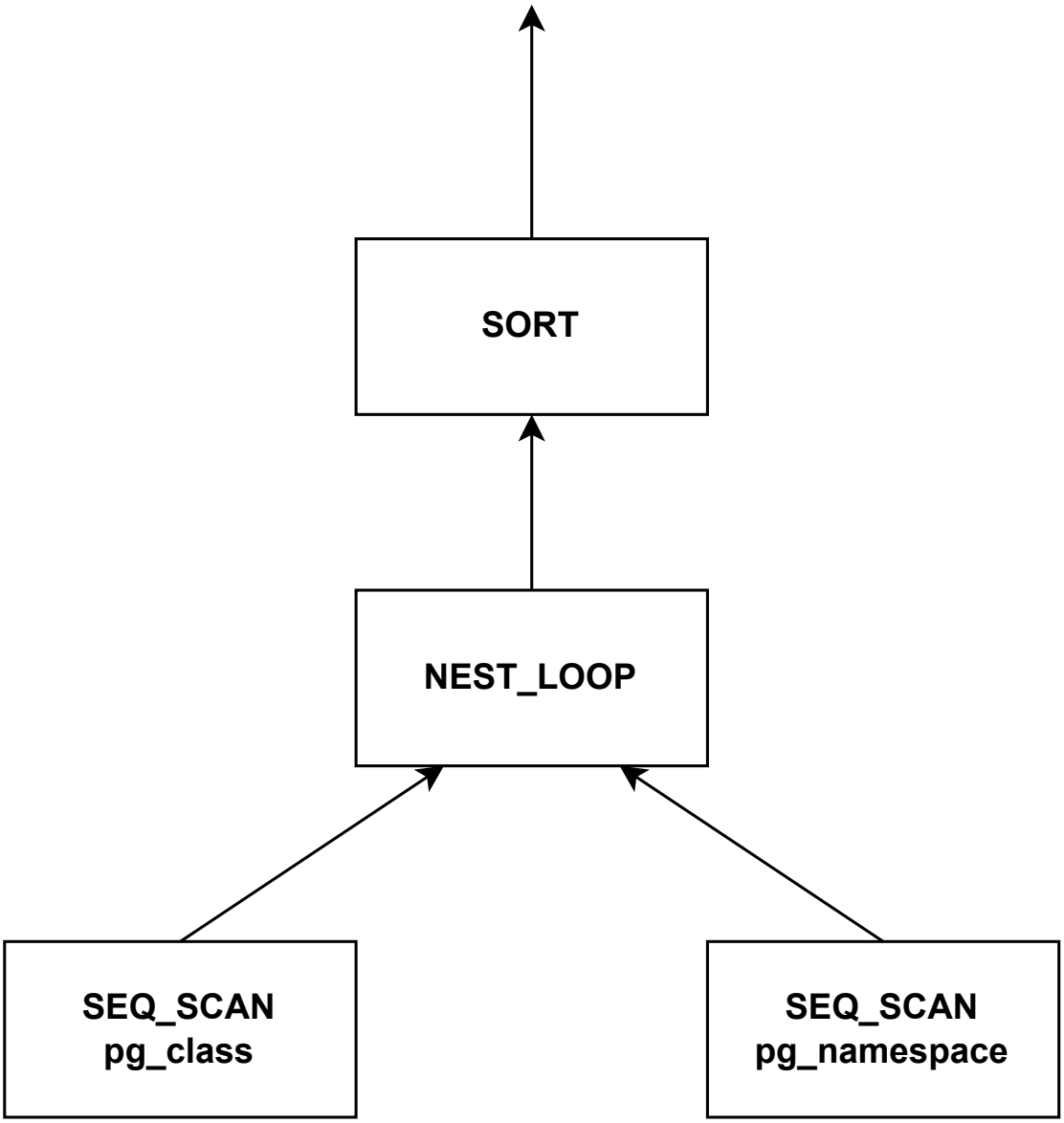
The cost is the planner's best estimate. Any planning mistakes will affect how much the cost will correlate with the actual time to execute. The primary purpose of cost assessment is to allow the planner to compare different execution plans for the same query in the same conditions. In any other case, comparing queries (worse, different queries) by cost is pointless and wrong. For example, consider a cost that was underestimated because the statistics were inaccurate. Update the statistics—and the cost may change, but the estimate will become more accurate, and the plan will ultimately improve.

2.6.3 Execution

An optimized query is executed in accordance with the plan.

An object called a portal is created in backend memory. The portal stores the state of the query as it is executed. This state is represented as a tree, identical in structure to the plan tree.

The nodes of the tree act as an assembly line, requesting and delivering rows to each other.



Execution starts at **the root node**. The root node (the sorting node `SORT` in the example)

requests data from the child node. When it receives all requested data, it performs the sorting operation and then delivers the data upward, to the client.

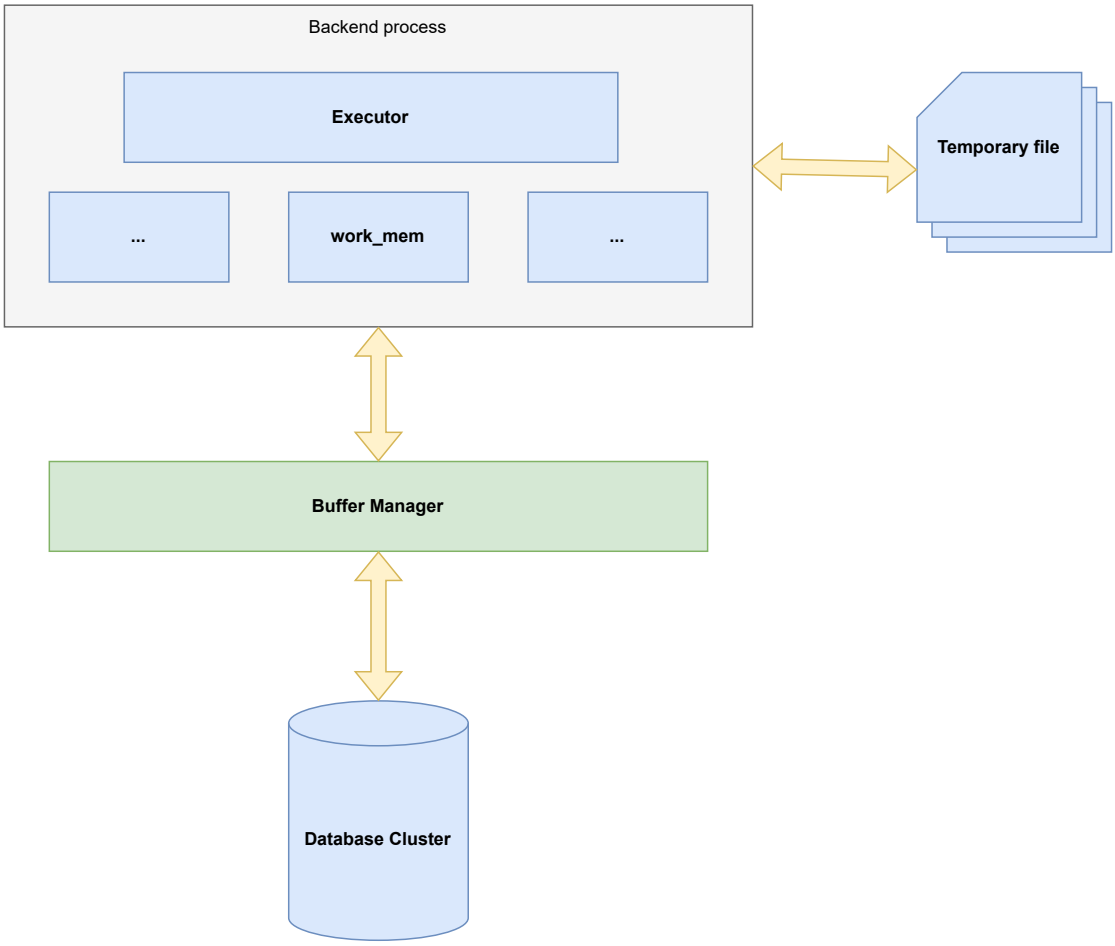
Some nodes (such as the `NEST_LOOP` node) join data from different sources. This node requests data from two child nodes. Upon receiving two rows that match the join condition, the node immediately passes the resulting row to the parent node (unlike with sorting, which must receive all rows before processing them). The node then stops until its parent node requests another row. Because of that, if only a partial result is required (as set by `LIMIT`, for example), the operation will not be executed fully. The two `SEQ_SCAN` leafs are table scans. Upon request from the parent node, a leaf node reads the next row from the table and returns it.

This node and some others do not store rows at all, but rather just deliver and forget them immediately. Other nodes, such as sorting, may potentially need to store vast amounts of data at a time. To deal with that, a `work_mem` memory chunk is allocated in backend memory. Its default size sits at a conservative 4MB limit; when the memory runs out, excess data is sent to a temporary file on-disk.

A plan may include multiple nodes with storage requirements, so it may have several chunks of memory allocated, each the size of `work_mem`. There is no limit on the total memory size that a query process may occupy.

A plan tree is composed of elements called plan nodes, and it is connected to the plantree list of the `PlannedStmt` structure. These elements are defined in `plannodes.h`. Each plan node has information that the executor requires for processing, and the executor processes from the end of the plan tree to the root in the case of a single-table query.

The executor reads and writes tables and indexes in the database cluster via the buffer manager. When processing a query, the executor uses some memory areas, such as `temp_buffers` and `work_mem`, allocated in advance and creates temporary files if necessary. The relationship among the executor, buffer manager and temporary files is illustrated in the following figure:



2.7 Query Optimization

2.7.1 Data Access Algorithms

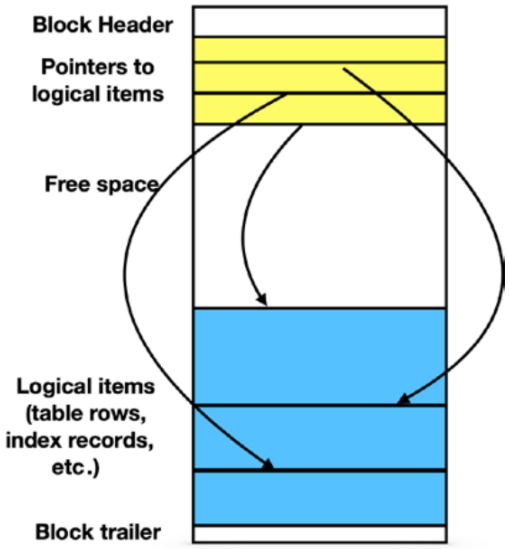
To begin executing a query, the database engine must extract stored data. This section concerns algorithms used to read data from database objects. In practice, these operations are often combined with their following operation in the query execution plan. This is advantageous in cases where it is possible to save execution time by avoiding reading that will be subsequently filtered out.

The efficiency of such operations depends on the ratio of rows that are retained to the total rows in the stored table. This ratio is called selectivity. The choice of algorithm for a given read operation depends on the selectivity of filters that can be simultaneously applied.

2.7.1.a Storage Structures

It should come as no surprise that data is stored in files that reside on hard drives. Any file used for database objects is divided in blocks of the same length; by default, PostgreSQL uses blocks containing 8192 bytes each. A block is the unit that is transferred between the hard drive and the main memory, and the number of I/O operations needed to execute any data access is equal to the number of blocks that are being read or written.

Database objects consist of logical items (table rows, index records, etc.). PostgreSQL allocates space for these items in blocks. Several small items can reside in the same block; larger items may spread among several blocks. The generic structure of a block is shown in Figure below.



The allocation of items to blocks also depends on the type of the database object. Table rows are stored using a data structure called a heap: a row can be inserted in any block that has sufficient free space, without any specific ordering. Other objects (e.g., indexes) may use blocks differently.

2.7.1.b Full Scan

In a full scan, the database engine consecutively reads all of the rows in a table and checks the filtering condition for each row. To estimate the cost of this algorithm, we need a more detailed description, as shown in the `pseudocode` below.

```
1 FOR each block IN a_table LOOP
2     read block;
3     FOR each row IN block LOOP
4         IF filter_condition (row)
```

```
        THEN output (row)
        END IF;
    END LOOP;
END LOOP;
```

The number of I/O accesses is BR; the total number of iterations of the inner loop is TR. We also need to estimate the cost of operations producing the output. This cost depends on selectivity, denoted as S, and is equal to S * TR. Putting all these parts together, we can estimate the cost of a full scan as

```
c1 * BR + c2 * TR + c3 * S* TR
```

where constants c1, c2, and c3 represent properties of hardware.

A full scan can be used with any table; additional data structures are not needed. Other algorithms depend on the existence of indexes on the table, described in the following.

2.7.1.c Index-Based Table Access

Note that until we got to physical operations, we did not even mention data access algorithms. We do not need to "read" relations—they are abstract objects. If we follow the idea that relations are mapped to tables, there is no other way to retrieve data than to read the whole table into the main memory. How else will we know which rows of data contain which values? But relational databases wouldn't be such a powerful tool for data processing if we stopped there. All relational databases, including PostgreSQL, allow for building additional, redundant data structures, making data access dramatically faster than a simple sequential read.

These additional structures are called indexes.

How indexes are built will be covered later in this chapter; for now, we need to understand two facts about indexes. First, they are "redundant" database objects; they do not store any additional information that can't be found in the source table itself.

Second, indexes provide additional data access paths; they allow us to determine what values are stored in the rows of a table without actually reading the table—this is how index-based access works. And, as mentioned previously, this happens entirely invisibly to the application.

If a filtering condition (or conditions) is encapsulated by an index on a table, the index can be used to access data from that table. The algorithm extracts a list of pointers to blocks that contain rows with values satisfying the filtering condition, and only these blocks are read from the table.

To get a table row from a pointer, the block containing this row must be read. The underlying data structure of a table is a heap, that is, rows are stored unordered. Their order is not guaranteed, nor does it correspond to properties of the data. There are two separate physical operations used by PostgreSQL to retrieve rows via indexes: index scan and bitmap heap scan. In an index scan, the database engine reads each entry of the index that satisfies the filter condition and retrieves blocks in index order. Because the underlying table is a heap, multiple index entries might point to the same block. To avoid multiple reads of the same block, the bitmap

heap scan implemented in PostgreSQL builds a bitmap indicating the blocks that contain needed rows. Then all rows in these blocks are filtered. An advantage of the PostgreSQL implementation is that it makes it easy to use multiple indexes on the same table within the same query, by applying logical ANDs and ORs on the block bitmaps generated by each index.

The cost model of this algorithm is much more complex. Informally, it can be described this way: for small values of selectivity, most likely, all rows satisfying the filtering conditions will be located in different blocks and, consequently, the cost is proportional to the number of result rows. For larger values of selectivity, the number of processed blocks approaches the total number of blocks. In the latter case, the cost becomes higher than the cost of a full scan because resources are needed to access the index.

2.7.1.d Index-Only Scan

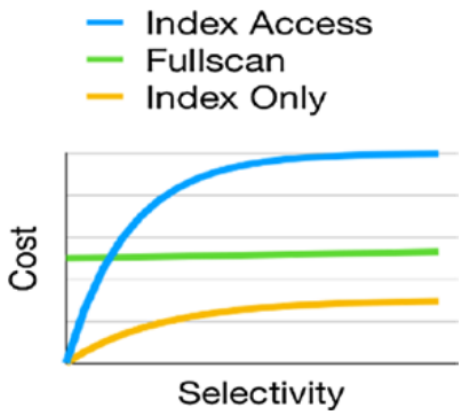
Data access operations do not necessarily return entire rows. If some columns are not needed for the query, these columns can be skipped as soon as a row passes filtering conditions (if any). More formally, this means that the logical project operation is combined with data access. This combination is especially useful if an index used for filtering contains all columns that are needed for the query.

The algorithm reads data from the index and applies remaining filtering conditions if necessary. Usually there is no need to access table data, but sometimes additional checks are needed.

The cost model for an index-only scan is similar to the model for index-based table access except that there's no need to actually access table data. For small values of selectivity, the cost is approximately proportional to the number of returned rows. For large values of selectivity, the algorithm performs an (almost) full scan of the index. The cost of an index scan is usually lower than the cost of a full table scan because it contains less data.

2.7.1.e Comparing Data Access Algorithms

The choice of the best data access algorithm depends mostly on query selectivity. The relationship of cost to selectivity for different data access algorithms is shown in Figure below. We intentionally omitted all numbers on this chart as they depend on hardware and table size, while the qualitative comparison does not.



The line for a full scan is linear and is almost horizontal because the growth is due to generation of output. Typically, the cost of output generation is negligible in comparison with other costs for this algorithm.

The line for a full scan is linear and is almost horizontal because the growth is due to generation of output. Typically, the cost of output generation is negligible in comparison with other costs for this algorithm.

The most interesting point is the intersection of two lines: for smaller values of selectivity, index-based access is preferable, while a full scan is better for larger values of selectivity. The position of the intersection depends on hardware and may depend on the size of the table. For relatively slow rotating drives, index-based access is preferable only if selectivity does not exceed 2–5%. For SSDs or virtual environments, this value can be higher. On older spinning disk drives, random block access can be an order of magnitude slower than sequential access, so the additional overhead of indexes is higher for a given proportion of rows.

The line representing an index-only scan is the lowest, meaning that this algorithm is preferable if it is applicable (i.e., all needed columns are in the index).

The query optimizer estimates both the selectivity of a query and the selectivity of the intersection point for this table and this index. The query shown below has a range filtering condition that selects a significant portion of the table.

A range filtering query executed with a full table scan

```
SELECT flight_no, departure_airport, arrival_airport
FROM flight
WHERE scheduled_departure BETWEEN
'2020-05-15' AND '2020-08-31';
```

In this case, the optimizer chooses a full scan

	QUERY PLAN text
1	Seq Scan on flight (cost=0.00..27058.64 rows=275729 width=12)
2	Filter: ((scheduled_departure >= '2020-05-15 00:00:00-05'::timestamp with time zone) AND (scheduled_departure <= '2020-08-...

However, a smaller range in the same query results in index-based table access. The query and its execution plan is shown below **A range filtering with index-based table access**

```
SELECT flight_no, departure_airport, arrival_airport
FROM flight
WHERE scheduled_departure BETWEEN
'2020-08-12' AND '2020-08-13';
```

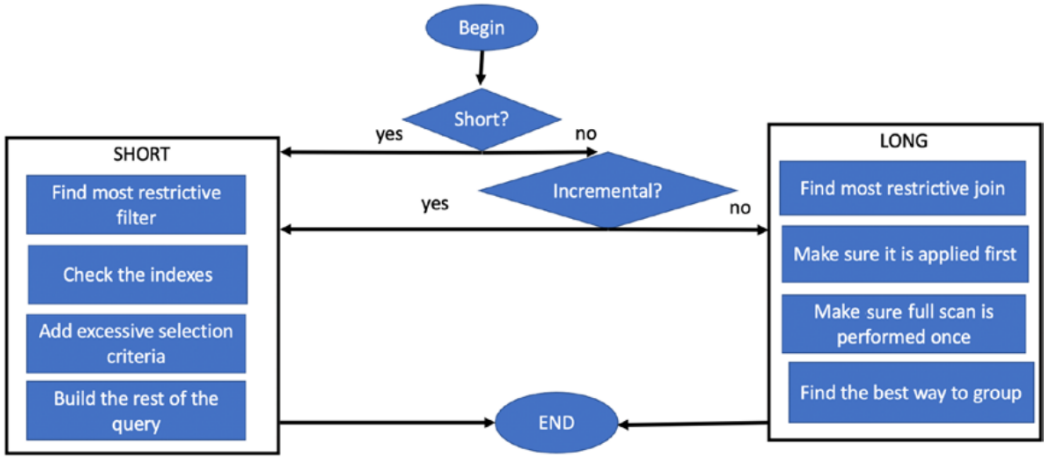
	QUERY PLAN text
1	Bitmap Heap Scan on flight (cost=83.27..9166.80 rows=3790 width=12)
2	Recheck Cond: ((scheduled_departure >= '2020-08-12 00:00:00-05'::timestamp w...
3	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..82.33 rows=37...
4	Index Cond: ((scheduled_departure >= '2020-08-12 00:00:00-05'::timestamp w...

In reality, the job of a query optimizer is much more complex: filtering conditions can be supported with multiple indexes with different values of selectivity. Multiple indexes can be combined to produce a block bitmap with fewer number of blocks to be scanned. As a result, the number of choices available to the optimizer is significantly larger than three algorithms.

Thus, there are no winners and losers among data access algorithms. Any algorithm can become a winner under certain conditions. Further, the choice of an algorithm depends on storage structures and statistical properties of the data. The database maintains metadata known as statistics for tables including information on things such as column cardinality, sparseness, and so on. Usually these statistics are not known during application development and may change throughout the application lifecycle. Therefore, the declarative nature of the query language is essential for system performance. More specifically, as the table statistics change or if other costing factors are adjusted, a different execution plan can be chosen for the same query.

2.7.2 Ultimate Optimization Algorithm

2.7.2.a Major Steps



2.7.2.b Step-by-Step Guide

STEP 1:
The first step is to determine whether the query in question is short or long. Looking at the query itself do not necessarily help you find the answer. Step 1 is a great time to recall that query optimization starts from gathering requirements, and it is important to work together with business owners and/or business analytics.

Check whether the business is interested in the most recent data or they need to follow historic trends, and so on. The business might say that they need to see all canceled flights, but it would be a good idea to ask whether they want to see all canceled flights from the beginning of time or within the past 24 hours.

If you determine that the query in question is short, go to Step 2; otherwise go, to Step 3.

STEP 2: Short
So, your query is a short query. Which steps do you need to follow to make sure that not only is it written in the best possible way but also that query performance will be stable even when data volumes grow?

- **STEP 2.1: The Most Restrictive Criteria**
  Find the most restrictive criteria for your query. Remember that often you can not tell which criteria this will be by just looking at the query. Query the tables to find the number of distinct values of attributes. Be aware of the value distribution (i.e., find out which values are the least frequent). When the most restrictive criteria are identified, proceed to the next step.
- **STEP 2.2: Check the Indexes**

In this step, you need to check whether there are indexes that support the search on the most restrictive condition. This includes the following:

  - Check whether all search attributes for the most restrictive condition are indexed. If the index(es) is missing, request or create one.
  - If more than one field is involved, check whether a compound index would perform better and whether the performance gains are enough to justify the creation of an additional index.
  - Check whether you can use an index-only scan using either a compound or covering index.

- **STEP 2.3: Add an Excessive Selection Criterion, If Applicable**
  If the most restrictive condition is based on a combination of attributes from different tables and thereby can't be indexed, consider adding an excessive selection criterion.

- **STEP 2.4: Constructing the Query**
  Start writing the query by applying the most restrictive criteria; this may mean starting from a select from a single table or a join that incorporates the most restrictive criteria. Do not omit this step. Often, when database developers know the relationships between objects, they tend to write all the joins before applying filtering. While we are aware that this is an often-recommended approach, we believe that for complex queries with multiple joins, it might complicate development. We suggest starting from a `SELECT` that you know is executed efficiently, and then adding one table at a time.

STEP 3: Long
Your query is a long query. In this case, the first step would be to determine whether you can use incremental refresh. Once again, this is when you need to work together with the business owner and/or business analysts to understand better what the purpose of the query is. Often, requirements are formulated without considering data dynamics. When the results of a query are stored in a table and it is updated periodically, it can either be pulled fresh each time (a full refresh, pulling all data from the dawn of time to the most recently available data), or it can be pulled incrementally, bringing in only data that has changed since the last data pull. The latter is what we mean by incremental updates. In the vast majority of cases, it is possible to pull data incrementally.

- If it is possible to use incremental updates, go to Step 4.
- Otherwise, go to Step 5.

STEP 4: Incremental Updates
Treat the query to select recently added or updated records as a short query with time of update being the most restrictive criterion. Go to Step 2 and follow the steps for optimizing short queries.

STEP 5:
If running incremental updates is not possible, proceed with the following steps of long query optimization:

- Find the most restrictive join, semi-join, or anti-join, if applicable, and make sure it is executed first.
- Keep adding tables to your join, one by one, and check the execution time and the execution plan each time.

- Make sure you do not scan any large tables multiple times. Plan your query to go through large tables only once.
- Pay attention to grouping. In the majority of cases, you need to postpone grouping to the last step, that is, you need to make sure that `GROUP BY` is the last statement in the execution plan. When grouping should be performed earlier to minimize the size of intermediate datasets.

3 HBase

We will discuss and present the necessary HBase database contents in this part.

3.1 Introduction to HBase

HBase, specifically Version 2.2.4, is a high-reliability, high-performance, column-oriented, scalable distributed storage system that uses HBase technology to build large-scale structured storage clusters on inexpensive PC Servers. The goal of HBase is to store and process large amounts of data, specifically to handle large amounts of data consisting of thousands of rows and columns using only standard hardware configurations.

Different from MapReduce's offline batch computing framework, HBase is random access storage and retrieval data platform, which makes up for the shortcomings of HDFS that cannot access data randomly.

It is suitable for business scenarios where real-time requirements are not very high - HBase stores Byte arrays, which do not mind data types, allowing dynamic, flexible data models.

To be able to study and learn HBase database, we mainly use resources, materials and knowledge written in [6, 4, 26] and [7].

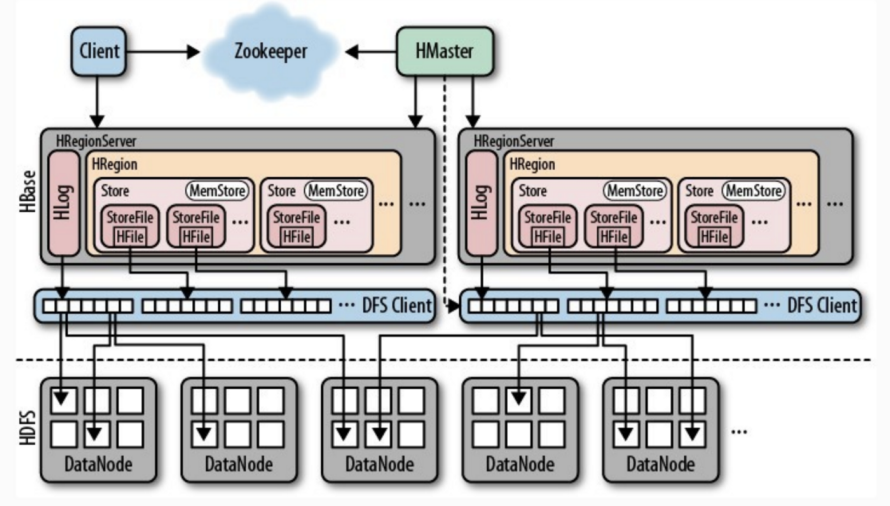
3.2 Data Storage and File Structures

3.2.1 HBase Architecture

HBase consists of **HMaster** and **HRegionServer** and also follows the **master-slave server architecture**. Typically, the HBase cluster has one Master node, called HMaster, and multiple Region Servers called HRegionServer. HMaster does not store any data itself, but only stores the mappings (metadata) of data to HRegionServer.

When the size of the table exceeds the preset value, HBase will automatically divide the table into different areas, each of which contains a subset of all the rows in the table. For the user, each table is a collection of data, distinguished by a primary key (RowKey). Physically, a table is split into multiple blocks, each of which is an HRegion. We use the table name + start/end primary key to distinguish each HRegion. One HRegion will save a piece of continuous data in a table. A complete table data is stored in multiple HRegions. Each Region Server contains multiple Regions – HRegions.

All nodes in the cluster are coordinated by Zookeeper and handle various issues that may be encountered during HBase operation. The basic architecture of HBase is shown below:



HBase Architecture

Client: Use HBase's RPC mechanism to communicate with HMaster and HRegionServer, submit requests and get results. For management operations, the client performs RPC with HMaster. For data read and write operations, the client performs RPC with HRegionServer.

Zookeeper: By registering the status information of each node in the cluster to ZooKeeper, HMaster can sense the health status of each HRegionServer at any time, and can also avoid the single point problem of HMaster.

The **HMaster** in the HBase is responsible for

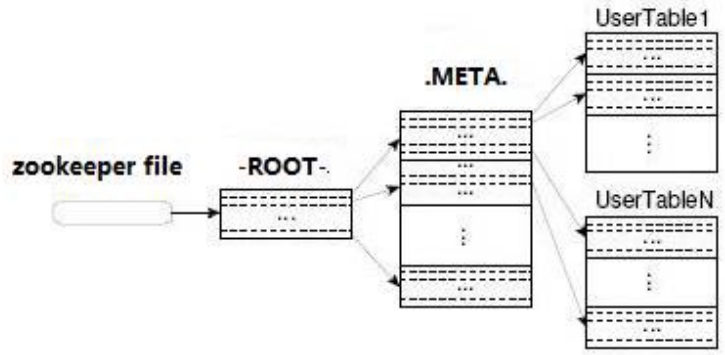
- Performing Administration.
- Managing and Monitoring the Cluster.
- Assigning Regions to the Region Servers.
- Controlling the Load Balancing and Failover.

On the other hand, the **HRegionServer** performs the following work

- Hosting and Managing Regions.
- Splitting the Regions automatically.
- Handling the read/write requests.
- Communicating with the Clients directly.

Each Region Server contains a Write-Ahead Log (called HLog) and multiple Regions. Each Region in turn is made up of a MemStore and multiple StoreFiles (HFile). The data lives in these StoreFiles in the form of Column Families (explained below). The MemStore holds in-memory modifications to the Store (data).

All HRegion metadata of HBase is stored in the .META. table. As HRegion increases, the



.META

data in the .META table also increases and splits into multiple new HRegions.

To locate the location of each HRegion in the .META table, the metadata of all HRegions in the .META the table is stored in the -ROOT-table, and finally, the location information of the ROOT-table is recorded by Zookeeper.

Before all clients access user data, they need to first access Zookeeper to obtain the location of -ROOT-, then access the -ROOT-table to get the location of the .META table, and finally determine the location of the user data according to the information in the META table, as follows: The -ROOT-table is never split. It has only one HRegion, which guarantees that any HRegion can be located with only three jumps. To speed up access, all regions of the .META tables are kept in memory.

The client caches the queried location information, and the cache does not actively fail. Each Region is identified by the start key (inclusive) and the end key (exclusive). If the client still does not have access to the data based on the cached information, then ask the Region server for the relevant .META table to try to obtain the location of the data. If it still fails, ask where the .META table associated with the -ROOT-table is.

Finally, if the previous information is all invalid, the data of HRegion is relocated by ZooKeeper. So if the cache on the client is entirely invalid, you need to go back and forth six times to get the correct HRegion.

3.2.2 HBase Data Model

HBase is a distributed database similar to BigTable. It is sparse long-term storage (on HDFS), multi-dimensional, and sorted mapping tables. The index of this table is the row keyword, column keyword, and timestamp. HBase data is a string, no type.

The Data Model in HBase is designed to accommodate semi-structured data that could vary

in field size, data type, and columns. Additionally, the layout of the data model makes it easier to partition the data and distribute it across the cluster. The Data Model in HBase is made of different logical components such as Tables, Rows, Column Families, Columns, Cells, and Versions.

Row Key	Customer		Sales	
Customer Id	Name	City	Product	Amount
101	John White	Los Angeles, CA	Chairs	$400.00
102	Jane Brown	Atlanta, GA	Lamps	$200.00
103	Bill Green	Pittsburgh, PA	Desk	$500.00
104	Jack Black	St. Louis, MO	Bed	$1600.00

Column Families

Table

Tables – The HBase Tables are more like a logical collection of rows stored in separate partitions called Regions. As shown above, every Region is then served by exactly one Region Server. The figure above shows a representation of a Table.

Rows – A row is one instance of data in a table and is identified by a rowkey. Rowkeys are unique in a Table and are always treated as a byte[].

Column Families – Data in a row are grouped together as Column Families. Each Column Family has one more Columns and these Columns in a family are stored together in a low-level storage file known as HFile. Column Families form the basic unit of physical storage to which certain HBase features like compression are applied. Hence it's important that proper care be taken when designing Column Families in the table. The table above shows Customer and Sales Column Families. The Customer Column Family is made up of 2 columns – Name and City, whereas the Sales Column Families is made up of 2 columns – Product and Amount.

Columns – A Column Family is made of one or more columns. A Column is identified by a Column Qualifier that consists of the Column Family name concatenated with the Column name using a colon – example: columnfamily:columnname. There can be multiple Columns within a Column Family and Rows within a table can have varied numbers of Columns.

Cell – A Cell stores data and is essentially a unique combination of rowkey, Column Family and the Column (Column Qualifier). The data stored in a Cell is called its value and the data type is always treated as byte[].

Version – The data stored in a cell is versioned and versions of data are identified by the timestamp. The number of versions of data retained in a column family is configurable and this value by default is 3.

Another example -

Row Key	Time Stamp	Column "Contents:"	Column "anchor:"		Column "mime:"
"com.cnn.www"	t9		"anchor: cnnsi.com"	"CNN"	
	t8		"anchor: my.look.ca"	"CNN.com"	
	t6	"<html>..."			"text/html"
	t5	"<html>..."			
	t3	"<html>..."			

Another Example

Think of a table as a large mapping. You can locate specific data by row key, row key plus timestamp, or row key plus column (column family: column modifier). Since HBase is sparsely storing data, some columns can be blank. The above table gives the logical storage logical view of the “com.cnn.www” website. There is only one row of data in the table.

The unique identifier of the row is “com.cnn.www”, and there is a time for each logical modification of this row of data. The stamp corresponds to the corresponding.

There are four columns in the table: contents: HTML, anchor:cnnsi.com, anchor:my.look.ca, mime: type and each column give the column family to which it belongs.

The row key (RowKey) is the unique identifier of the data row in the table and serves as the primary key for retrieving records.

There are only three ways to access rows in a table in HBase: access via a row key, range access for a given row key, and full table scan.

The row key can be any string (maximum length 64KB) and stored in lexicographical order. For rows that are often read together, the fundamental values need to be carefully designed so that they can be stored collectively.

3.3 Indexing

If you want to query a piece of data accurately in HBase, you must use rowkey. If you do not use rowkey to query data, you must compare row by row and column by column (i.e. full table scanning) as HBase does not have any built-in support for any other type of indexes. In actual business, you need to query data quickly through multiple dimensions. For example, when you query users, you may need to query through user name, name, email address, mobile phone number, However, it is obviously impossible to put such multi-dimensional query fields in rowkey due to the reduction in flexibility as well as the limitation in length of the rowkey.

In order to create indexes on fields other than the row key, we have two options: incorporating the use of another software, such as the Apache Phoenix engine, or leveraging a useful features in HBase like co-processors to create and manage the index ourselves, both of which will be discussed in this report.

3.3.1 Coprocessor in HBase

Simply stated, Coprocessor is a framework that provides an easy way to run custom code on Region Server. It is a framework inspired by Google's BigTable coprocessors [3], designed for both flexible and generic extension, and of distributed computation directly within the HBase server processes. Some characteristic of coprocessor:

- Arbitrary code can run at each tablet in table server
- High-level call interface for clients Calls are addressed to rows or ranges of rows and the coprocessor client library resolves them to actual locations; Calls across multiple rows are automatically split into multiple parallelized RPC
- Provides a very flexible model for building distributed services
- Automatic scaling, load balancing, request routing for applications

Since it is built for extension, HBase can easily integrate new and exiting new features on top of it such as secondary indexing, complex filtering (push down predicates), and access control. Coprocessors can be loaded globally on all tables and regions hosted by the region server, these are known as system coprocessors; or the administrator can specify which coprocessors should be loaded on all regions for a table on a per-table basis, these are known as table coprocessors.

In order to support sufficient flexibility for potential coprocessor behaviors, two different aspects of extension are provided by the framework. One is the observer, which are like triggers in conventional databases, and the other is the endpoint, dynamic RPC endpoints that resemble stored procedures.

3.3.1.a Observers

The idea behind observers is that we can insert user code by overriding upcall methods provided by the coprocessor framework. The callback functions are executed from core HBase code when certain events occur. The coprocessor framework handles all of the details of invoking callbacks during various base HBase activities; the coprocessor need only insert the desired additional or alternate functionality.

The current version of observers coprocessor in HBase provides these interfaces:

- RegionObserver: Provides hooks for data manipulation events, Get, Put, Delete, Scan, and so on. There is an instance of a RegionObserver coprocessor for every table region and the scope of the observations they can make is constrained to that region.
- WALObserver: Provides hooks for write-ahead log (WAL) related operations. This is a way to observe or intercept WAL writing and reconstruction events. A WALObserver runs in the context of WAL processing. There is one such context per region server.
- MasterObserver: Provides hooks for DDL-type operation, i.e., create, delete, modify table, etc. The MasterObserver runs within the context of the HBase master.
- Region Server Observer: Provides hook for the events related to the RegionServer, such as stopping the RegionServer and performing operations before or after merges, commits, or rollbacks.

Here are some callbacks that those interfaces - RegionObserver - provide:

- preOpen, postOpen: Called before and after the region is reported as online to the master.
- preFlush, postFlush: Called before and after the memstore is flushed into a new store file
- preGet, postGet: Called before and after a client makes a Get request.
- preExists, postExists: Called before and after the client tests for existence using a Get.
- prePut and postPut: Called before and after the client stores a value.
- preDelete and postDelete: Called before and after the client deletes a value.

For ease of implementation, all of these methods are implemented with default behaviors in the abstract class BaseRegionObserver so you can focus on what events you have interest in, without having to be concerned about process upcalls for all of them. Here is a sequence diagram that shows how RegionObservers works with other HBase components:

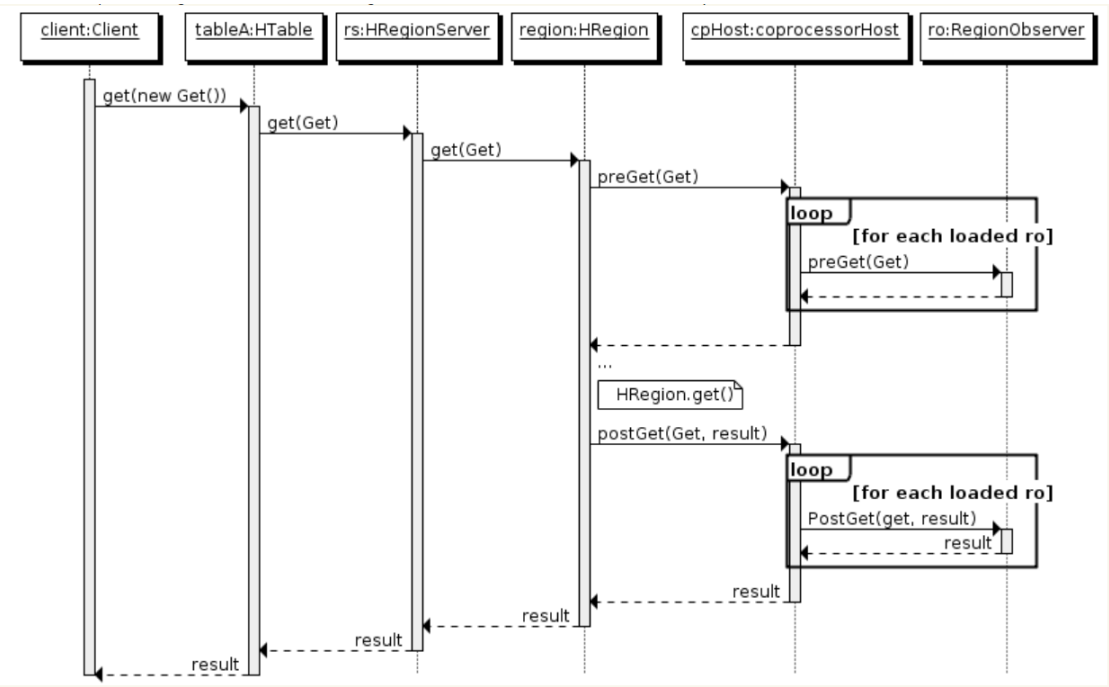


Figure 1: RegionObserver's sequence diagram [11]

The MasterObserver interface provides upcalls for:

- preCreateTable/postCreateTable: Called before and after the region is reported as online to the master.
- preDeleteTable/postDeleteTable

WALObserver provides upcalls for:

- preWALWrite/postWALWrite: called before and after a WALEdit written to WAL.

3.3.1.b Endpoint

As mentioned previously, observers can be thought of like database triggers. Endpoints, on the other hand, are more powerful, resembling stored procedures. One can invoke an endpoint at any time from the client. The endpoint implementation will then be executed remotely at the target region or regions, and results from those executions will be returned to the client.

Endpoint is an interface for dynamic RPC extension. The endpoint implementation is installed on the server side and can then be invoked with HBase RPC. The client library provides convenience methods for invoking such dynamic interfaces.

Also as mentioned above, there is nothing stopping the implementation of an endpoint from communicating with any observer implementation. With these extension surfaces combined you can add whole new features to HBase without modifying or recompiling HBase itself. This can be very powerful.

In order to build and use your own endpoint, you need to:

- Have a new protocol interface which extends CoprocessorProtocol.
- Implement the Endpoint interface. The implementation will be loaded into and executed from the region context.
- Extend the abstract class BaseEndpointCoprocessor. This convenience class hides some internal details that the implementer need not necessary be concerned about, such as coprocessor framework class loading.
- On the client side, the Endpoint can be invoked by two new HBase client APIs:
  - Executing against a single region:

```
HTableInterface.coprocessorProxy(Class<T> protocol, byte[] row)
```

  - Executing over a range of regions:

```
HTableInterface.coprocessorExec(Class<T> protocol, byte[] startKey, byte[]
endKey, Batch.Call<T,R> callable)
```

3.3.2 Indexing with Coprocessor

As mentioned before, when you want to index something that is not the rowkey, you will have to manage it youself in HBase, and coprocessor is a helpful feature that can help you to maintain and synchronize your index table and the data table. Here, we will use the RegionObserver coprocessor class to implement a secondary index by overriding the postPut method.

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.Cell;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;
```

```
import org.apache.hadoop.hbase.client.Durability;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.coprocessor.BaseRegionObserver;
import org.apache.hadoop.hbase.coprocessor.ObserverContext;
import org.apache.hadoop.hbase.coprocessor.RegionCoprocessorEnvironment;
import org.apache.hadoop.hbase.regionserver.wal.WALEdit;

public class SecondaryIndex extends BaseRegionObserver{
    static Configuration conf = null;
    static Connection conn = null;
    static HTable table = null;
    static {
        try {
            conf = HBaseConfiguration.create();
            conf.set("hbase.zookeeper.quorum",
            "hadoop01:2181,hadoop02:2181,hadoop03:2181");
            conn = ConnectionFactory.createConnection(conf);
            table = (HTable) conn.getTable(TableName.valueOf("mingxing_user"));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    /*

  * Method for calling when the trigger
         * After inserting the data, return to the client to intercept the inserted
  data to specify the operation.
            * Insert the data to the index table
            * Parameter 1: context object environment variable
            * Parameters 2: PUT object coprocessor interception PUT object
            * Insert the data of the original table
            * Parameters 3: Life Cycle
            * To user_mingxing
     * put "user_mingxing","rk001","info:name","luhan"
       */
       @Override
       public void postPut(ObserverContext<RegionCoprocessorEnvironment> e,
       Put put, WALEdit edit, Durability durability)
        throws IOException {
       //Need to analyze objects, original table objects
       byte[] row = put.getRow();
       Cell cv = put.get("info".getBytes(), "name".getBytes()).get(0);
       byte[] value = cv.getValue();
        //Data inserting a new table
       //Package the object RK V to the Mingixng_User Table RK V
       Put newput =new Put(value);
       newput.add("info".getBytes(), "fensi".getBytes(), row);
       table.put(newput);
       table.close();
       }

}
```

From the example code above, we create a secondary index as the class SecondaryIndex which extends from the BaseRegionObserver class. The class has an **HTable** attribute to represent a table and a **postPut** method which play a similar role to a trigger in relational DBMS, performing certain action after a **PUT** command is executed. When the **postPut** method is called, it will intercept the inserted data and put the necessary information into the secondary index table.

In term of index type and structure, this is a secondary index, so we have to create a new index entry for each new record, and since this index is stored in a separate table, there is no special structure to support index like B+ tree, it is just another HBase table.

While implementing secondary index with coprocessor is possible, it is not always the best option due to its various drawbacks:

- Increased overhead: since the user have to both implement and maintain the index table themselves, it added a significant amount of overhead in term of complexity, cost and time,
- It is functionally limited compared to others solutions. Others option to implement secondary index in HBase can create index on functions, while using coprocessor is limited to index only on the data itself.

3.3.3 Index with Apache Phoenix

3.3.3.a Apache Phoenix overview

Due to limitations in the way secondary index is implemented using coprocessor in HBase, other solutions has been developed with improvement to both ease of use and functionality in mind, and Apache Phoenix is one of the most successful alternatives.

Apache Phoenix is a relational database layer over HBase delivered as a client-embedded JDBC driver targeting low latency Queries over HBase data. Apache Phoenix takes your SQL versioned, such that snapshot queries over prior versions would automatically use the correct Performance on the order of milliseconds for small queries, or seconds for tens of millions of rows.

Phoenix acts as an inline client Java database connectivity (JDBC) driver for low-latency access to data in HBase. Apache Phoenix compiles a user-written SQL query into a series of scan operations, resulting in a generic JDBC result set returned to the client. The metadata for the data table is stored in the HBase table and is marked with the version number, so the correct schema is automatically selected when the query is made. Using HBase APIs directly, combined with coprocessor (coprocessor) and custom filters, small-scale queries respond in milliseconds, and tens of millions of data respond in seconds.

The Phoenix structure is divided into two parts: the client and the server.

The client includes application development, analyzing SQL to generate QueryPlan, which is converted to HBase Scans, call HBase API to issue query computing requests, and receive return results; The server mainly uses HBase's coprocessor, processing secondary index, aggregation, and JOIN calculations.

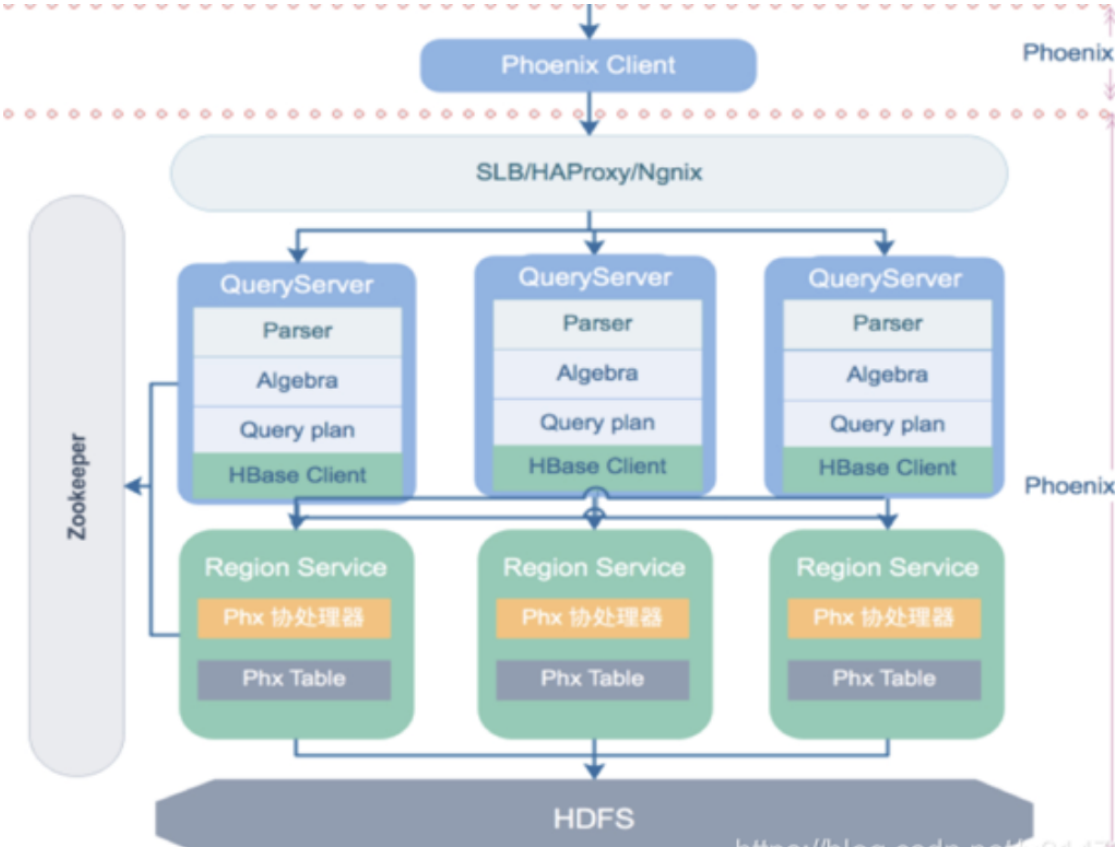


Figure 2: Phoenix's architecture [8]

The client is the smallest JDBC driver for users, decoupled with Phoenix depends, supports multiple language clients such as Java, Python, Go, which is a huge improvement in term of language compatibility for HBase.

Deploy QueryServer as a separate HTTP service to receive light client requests, analyze, optimize, and generate execution plans for SQL.

3.3.3.b Indexing in Phoenix

There are two special kind of index in Phoenix, which are described in this section.

Functional Index

Built on functions rather than just columns [25]. It allow you to create an index not just on columns, but on an arbitrary expressions. Then when a query uses that expression, the index may be used to retrieve the results instead of the data table.

```
1    CREATE INDEX UPPER_NAME_IDX ON EMP (UPPER(FIRST_NAME||' '||LAST_NAME))
```

With this index in place, when the following query is issued, the index would be used instead of the data table to retrieve the results:

```
SELECT EMP_ID FROM EMP WHERE UPPER(FIRST_NAME||' '||LAST_NAME)='JOHN DOE'
```

Covered Indexes

Covered index is a way to bundle data based on alternative access path. If the index can "cover" all fields in your select statement then only the index will be hit during the query. We do not need to go back to the primary table once we have found the index entry. Instead, we bundle the data we care about right in the index rows, saving read-time overhead.

For example, the following would create an index on the v1 and v2 columns and include the v3 column in the index as well to prevent having to get it from the data table:

```
CREATE INDEX my_index ON my_table (v1,v2) INCLUDE(v3)
```

Apart from those two index types, Phoenix supports two types of indexing techniques: global and local indexing. Each are useful in different scenarios and have their own failure profiles and performance characteristics.

- Global indexing: suitable for read heavy scenarios [1]. Using global indexing is expensive when data is written frequently, because all the update operations (delete, update values and upgrade select) of the data table will cause the update of the index table, and the index table is distributed on different data nodes, and the cross node data transmission brings great performance consumption. When reading data, Phoenix selects the index table to reduce the query time. By default, if the field to be queried is not an index field, the index table will not be used, that is to say, it will not improve the query speed.
- Local indexing, which is suitable for write heavy scenarios [1]. Like global indexing, Phoenix automatically determines whether to use the index when making queries. When using local indexing, the index data and the data of the data table are stored in the same server, which avoids the extra cost of writing indexes to the index tables of different servers during the write operation. When using local indexing, even if the query field is not an index field, the index table will be used, which will improve the query speed, which is different from Global indexing. For local indexing, all index data of a data table is stored in a single, independent and sharable table.

3.4 Query Processing

HBase is optimized for reads when data is queried on basis of row key. Query predicates are applied to row key as start and stop keys. Reading data from HBase on the basis of rowkey is somewhat similar to using index for reading a RDBMS table [7].

- To retrieve particular RowKey records:

```
        scan 'TABLENAME',{FILTER =>"(PrefixFilter ('ROWKEY'))"}
```

- To retrieve particular RowKey with limit:

```
scan 'TABLENAME',{LIMIT => 10, FILTER =>"(PrefixFilter ('ROWKEY'))"}
```

- To list only those rows, whose row keys have a particular word in it:

```
scan 'TableName', {FILTER =>
org.apache.hadoop.hbase.filter.RowFilter.new(CompareFilter::CompareOp
.valueOf('EQUAL'),SubstringComparator.new("search
word"))}
```

However, if you have to perform huge scans and joins between big tables, you can use the **Apache Phoenix** to run HBase queries in **Azure HDInsight** as HBase has no its build-in query language. Apache Phoenix is accessed as a JDBC driver, and it enables querying and managing HBase tables by using SQL. SQLLine is a command-line utility to execute SQL.

In this section, we will illustrate the process of connecting and querying HBase data using Apache Phoenix.

Here are some prerequisites:

- An Apache HBase cluster. A tutorial of creating an Apache HBase cluster in Azure HDInsight can be found in this link.
- An SSH client. Using SSH to securely connect to Apache Hadoop on Azure HDInsight.

After connecting to HBase cluster, user need to connect to one of the Apache ZooKeeper nodes. Here is an example command line with corresponding output:

```
curl -u admin:PASSWORD -sS -G https://CLUSTERNAME.azurehdinsight
.net/api/v1/clusters/CLUSTERNAME/services/ZOOKEEPER/components/ZOOKEEPER_SERVER
```

```
{
  "href" : "http://hn*.432dc3rlshou3ocf251eycoapa.bx.internal.cloudapp
  .net:8080/api/v1/clusters/myCluster/hosts/<zookeepername1>
  .432dc3rlshou3ocf251eycoapa.bx.internal.cloudapp
  .net/host_components/ZOOKEEPER_SERVER",
  "HostRoles" : {
    "cluster_name" : "myCluster",
    "component_name" : "ZOOKEEPER_SERVER",
    "host_name" :
    "<zookeepername1>.432dc3rlshou3ocf251eycoapa.bx.internal.cloudapp.net"
  }
```

Now, user can start connecting to HBase clusters using SSH, and then use the Apache Phoenix to create HBase tables, insert data, and query data.

Set up the working space -

- Using SSH command to connect to HBase cluster.

```
ssh sshuser@CLUSTERNAME-ssh.azurehdinsight.net
```

- Then, user need to change the directory to the Phoenix client using the following command:

```
cd /usr/hdp/current/phoenix-client/bin
```

- Launch the SQLLine, a pure-Java console based utility for connecting to relational databases and executing SQL commands.

```
./sqlline.py ZOOKEEPER:2181:/hbase-unsecure
```

Example with HBase -

- Create an HBase table:

```
CREATE TABLE Company (company_id INTEGER PRIMARY KEY, name VARCHAR(225));
```

- Show list of tables:

```
!tables
```

```
+------------+--------------+-------------------------------+-------------+
| TABLE_CAT  | TABLE_SCHEM  |           TABLE_NAME          | TABLE_TYPE  |
+------------+--------------+-------------------------------+-------------+
|            | MYDB         | Company                       | TABLE       |
+------------+--------------+-------------------------------+-------------+
```

- Insert values into tables:

```
UPSERT INTO Company VALUES(1, 'Microsoft');
UPSERT INTO Company VALUES(2, 'Apache');
UPSERT INTO Company VALUES(3, 'Apple');
```

```
+------------+----------------------------------+
| company_id |              name                |
+------------+----------------------------------+
|1           | Microsoft                        |
+------------+----------------------------------+
|2           | Apache                           |
+------------+----------------------------------+
|3           | Apple                            |
+------------+----------------------------------+
```

- Perform query the table:

```
SELECT * FROM Company WHERE company_id = 2;
```

```
+------------+---------------------------------+
| company_id |              name               |
+------------+---------------------------------+
|2           | Apache                          |
+------------+---------------------------------+
```

- Delete record from a table:

```
DELETE FROM Company WHERE COMPANY_ID=1;
```

- Drop a table in the database with the following command:

```
DROP TABLE Company;
```

In conclusion, HBase is most efficient for reading a single row or a range of rows. Wrong operation on rowkey may lead to entire table scanning, which is not an optimal solution. Also, when using external module to integrate with HBase, user need to consider carefully to optimize the performance of this NoSQL database.

3.5 Query Optimization

HBase supports several parameters for user's purpose. User can tune these parameters based on user's use case, and make the HBase work in an optimized way. In this section, our group will try to explain some the parameters that can be tuned for user's better experience [9].

- `hbase.regionserver.handler.count`
  This parameter defines the number of RPC listeners/threads that are kept open to answer incoming requests to user tables. The default value is 30. If more concurrent users are trying to access HBase, it is good to keep this value higher. Also, each thread consumes a significant amount memory and CPU cycles, therefore, it need to be proportional to number of CPU cores and region server in each region server.

  A rule of thumb is to keep the value low when the payload for each request is large, and keep the value high when the payload is small.

- `hbase.hregion.max.filesize`
  This parameter is the maximum HStoreFile size, the default value is 10GB. The rate at which regions in HBase split can practically be managed. The entire region would be separated once "any" store (Column family) inside it reaches this value.

- `hbase_master_heapsize`
  A master in HBase seldom needs more than 4 to 8 GB in typical installations because it does not perform any heavy lifting. Master is basically responsible for meta operations such as create/ delete of tables, monitoring the health of region servers via zookeeper znodes, and redistributing regions during startup *balancer* or when a region server goes down. Rhe assignment manager for the master only maintains track of region states in this memory, thus, if user has a huge number of of tables/regions `hbase_master_heapsize` need to be keep proportional to the amount of heap for master.

- `hbase_regionserver_heapsize`
  Most of the data loading/processing happens in allocated server heap and `hbase_region server_heapsize` is the heap memory accommodating block cache to make the data processing faster and holding all the writes coming from your users (until they get flushed to the disk).

- `file.block.cache.size`
  Percentage of maximum heap (-Xmx setting) to allocate to block cache used by a StoreFile. Default of 0.4 means allocate 40%. When a user requests the same data again, it is provided from this cache rather than directly from the disk, which is much faster. The data that was previously retrieved from disk is loaded into this cache.

- `hbase.regionserver.global.memstore.size`
  Maximum size of all memstores in a region server before new updates are blocked and flushes are forced. Defaults to 40% of heap. In order to ensure that we have adequate heap available for standard hbase operations in addition to read and write caching, take note that the sum of the block cache as stated in point 4 above and the size of the global memstore should never be larger than 70 to 75% of the entire heap.

- `hbase.hregion.memstore.flush.size`
  Memstore will be flushed to disk if size of the memstore exceeds this number of bytes. Value is checked by a thread that runs every `hbase.server.thread.wakefrequency`. Each flush operation generates a hfile, therefore the smaller this number, the likelihood of having more frequent flushes, more IO overhead, the creation of more hfiles, and ultimately, the faster the onset of compaction. With the caveat that the total heap size and the number of regions and column families on each region server must be taken into account, a substantially larger flush size would guarantee fewer Hfiles and smaller compactions. You cannot afford to have a larger flush size under a constrained total heap size if you have an excessive number of regions and column families. Any amount between 128 MB and 256 MB is ideal.

- `hbase.hstore.blockingStoreFiles`
  If more than this number of StoreFiles exist in any one Store (one StoreFile is written per flush of MemStore), updates are blocked for this region until a compaction is completed, or until `hbase.hstore.blockingWaitTime` has been exceeded. When this limit is reached, the region server for this region will stop accepting writes, and you will see messages like `"org.apache.hadoop.hbase.RegionTooBusyException: Above memstore limit"` in the logs. Once minor compaction is over, the situation will be under control and writing will restart. To avoid any potential problems with such high write traffic, one can always increase this setting, which could also make channels more effective. For added assistance, one may additionally raise the value of `hbase.hstore.compaction.max` to ensure that more Hfiles are included in the compaction process.

- `hbase.hstore.compaction.max`
  The maximum number of StoreFiles which will be selected for a single minor compaction, regardless of the number of eligible StoreFiles. Effectively, the value of `hbase.hstore.compaction.max` controls the length of time it takes a single compaction to complete. Setting it larger means that more StoreFiles are included in a compaction. For most cases, the default value is appropriate.

- `hbase.hregion.max.filesize`
  Maximum file size. If the sum of the sizes of a region's HFiles has grown to exceed this

value, the region is split in two. There are two choices of how this option works, the first is when any store's size exceed the threshold then split, and the other is overall region's size exceed the threshold then split, it can be configed by hbase.hregion.split.overallfiles.

- `zookeeper.session.timeout`
  The timeout is 90 seconds by default (specified in milliseconds). As a result, if a server crashes, it will take the Master 90 seconds to discover the problem and initiate recovery. So that the Master discovers errors sooner, you might need to adjust the timeout to a minute or even less. Make sure your JVM garbage collection setup is in order before changing this value; otherwise, a long garbage collection that lasts longer than the ZooKeeper session timeout would destroy your RegionServer. (You might be okay with this; if a RegionServer has been in GC for a while, you probably want recovery to begin on the server.)

- `hbase.client.scanner.caching`
  If the scanner cannot be supplied from (local, client) memory, the number of rows that we attempt to fetch while calling next. To make the most of the network, this configuration cooperates with `hbase.client.scanner.max.result.size`. Since the size of rows varies from table to table, the default value of Integer.MAX VALUE ensures that the network will fill the chunk size specified by `hbase.client.scanner.max.result.size` rather than being constrained by a specific number of rows. If you know in advance that you do not need more than a specific number of rows from a scan, you should use `Scan#setCaching` to set this setting to that amount of rows. While keeping the count low underutilizes the cluster and scanner, keeping the count high heavily utilizes client RAM and the region server heap. Therefore, a good quantity depends on the cluster nodes' disk, memory, and CPU capabilities as well as the number of million rows that need to be scanned. If the demands are high, go high.

4 Application

In this assignment, we designed and developed a website to sell e-commerce online. We call this website BK Store. You can find out more about our application here [2].

4.1 Roles

There are two main roles on our website:

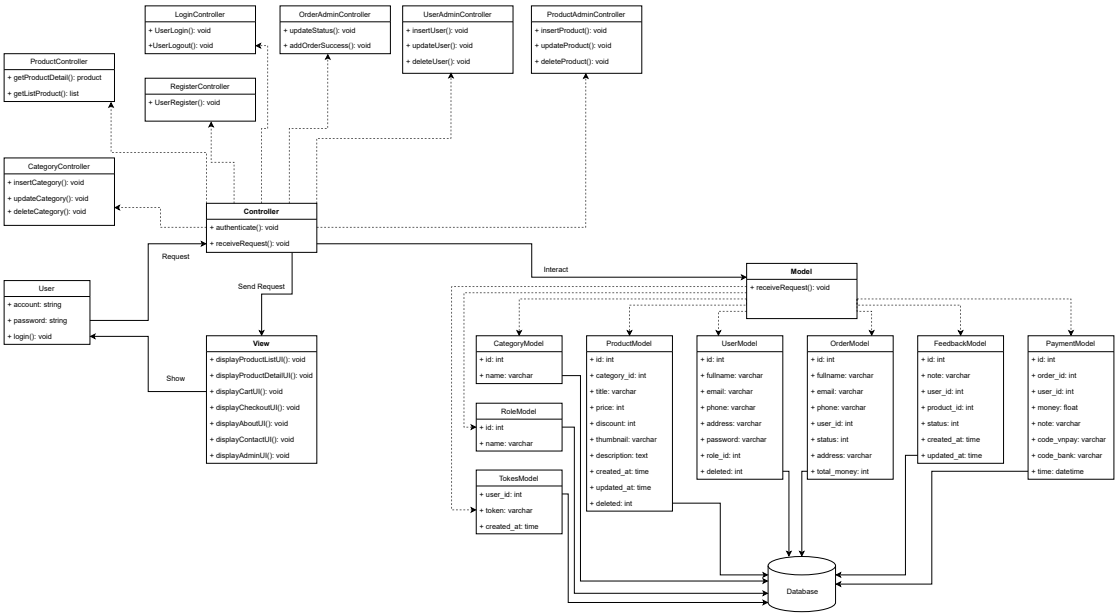
- **Admin**
  - This is the website administrator.
  - Can directly edit everything on the website including the interface, database, etc.
  - Can view the store's statistical data (revenue, sales, etc).
  - Can accept or reject customer orders.
  - Can edit orders or can create orders on behalf of customers.
  - Can send notifications to customers if problems arise.
  - Can hide a product on the product list.
- **User**
  - As a customer using the service at the store.
  - Log in to the system with the customer's account.
  - Create orders and edit orders according to customer preferences.
  - Can add, remove or edit quantity or products included in the order.
  - Provided and supported with various payment methods from domestic (domestic banks, e-wallets, etc) to international (VISA, PayPal, etc).

4.2 Architecture

Please note that all diagrams in this section are presented as **Portable Document Format (PDF)** files, so if you find them blurry or unclear, please zoom in for a better view.

4.2.1 MVC Model

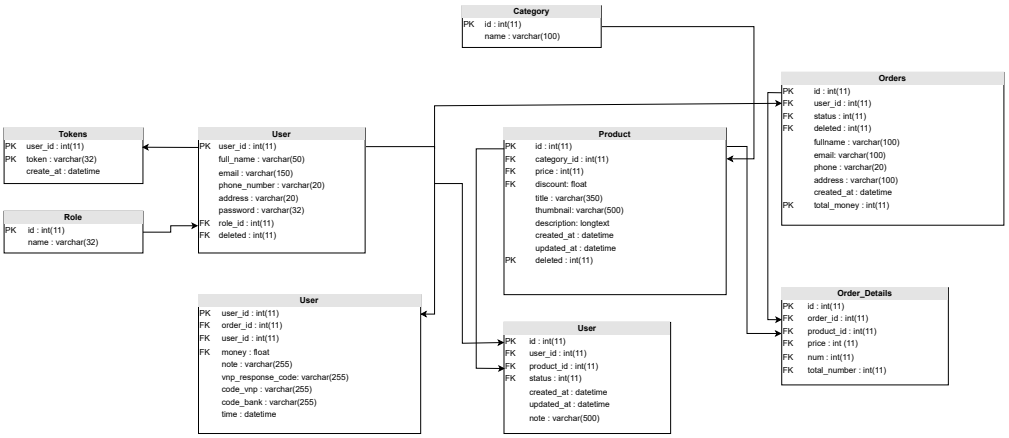
The Model-View-Controller (MVC) is an architectural pattern that separates an application into three main logical components: the model, the view, and the controller. Each of these components are built to handle specific development aspects of an application. MVC is one of the most frequently used industry-standard web development framework to create scalable and extensible projects. We use this model to build our application.



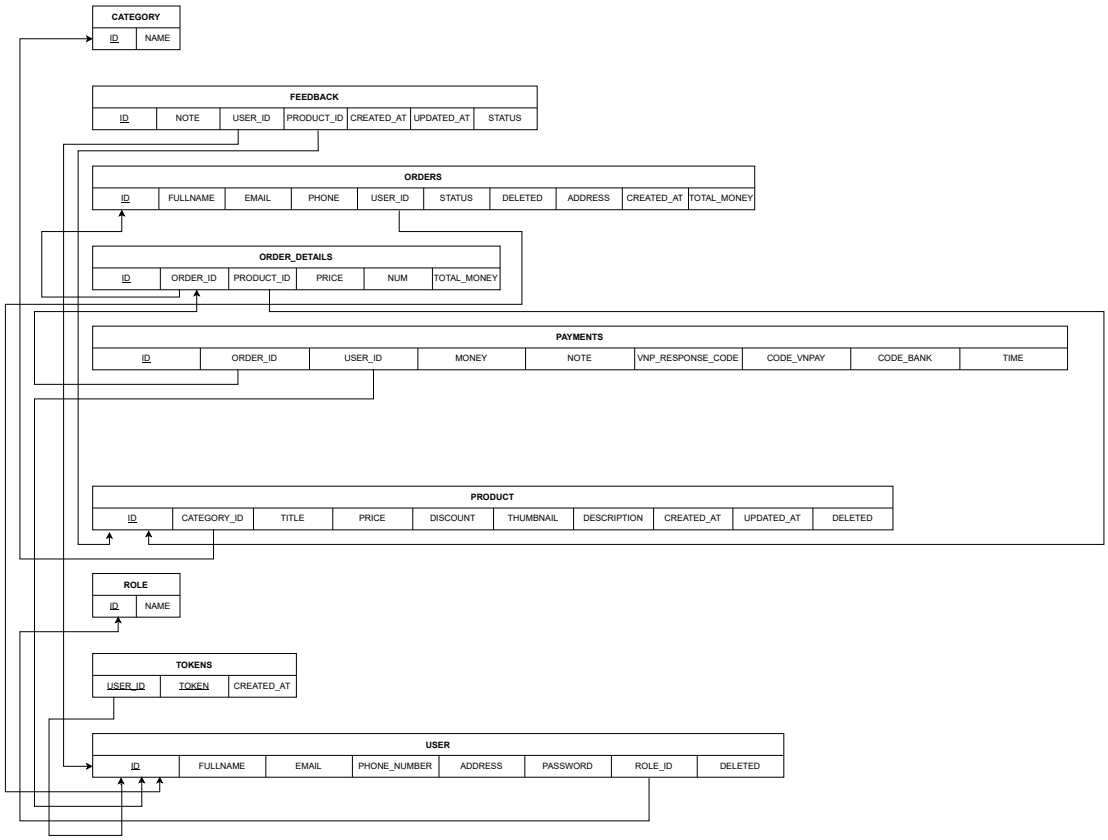
MVC Model Diagram

4.2.2 PostgreSQL Database Diagram

Since PostgreSQL is a relational database, it is important to have a database diagram for visualization.

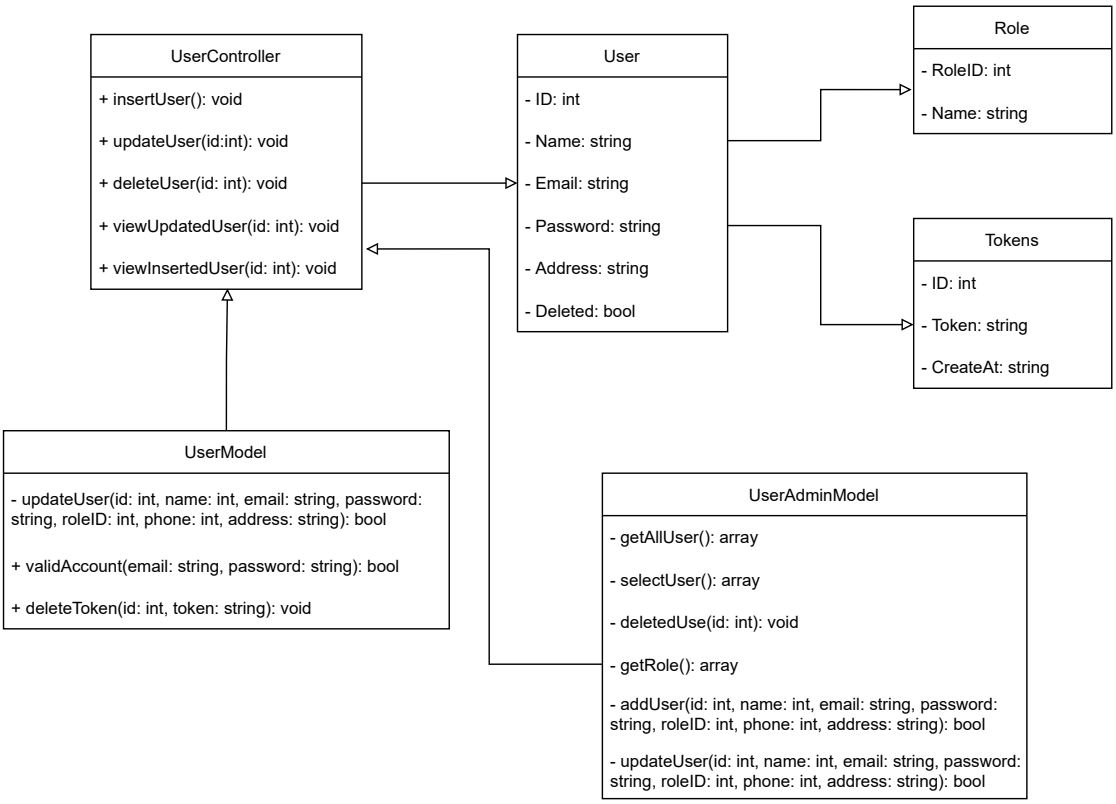


PostgreSQL Database Diagram

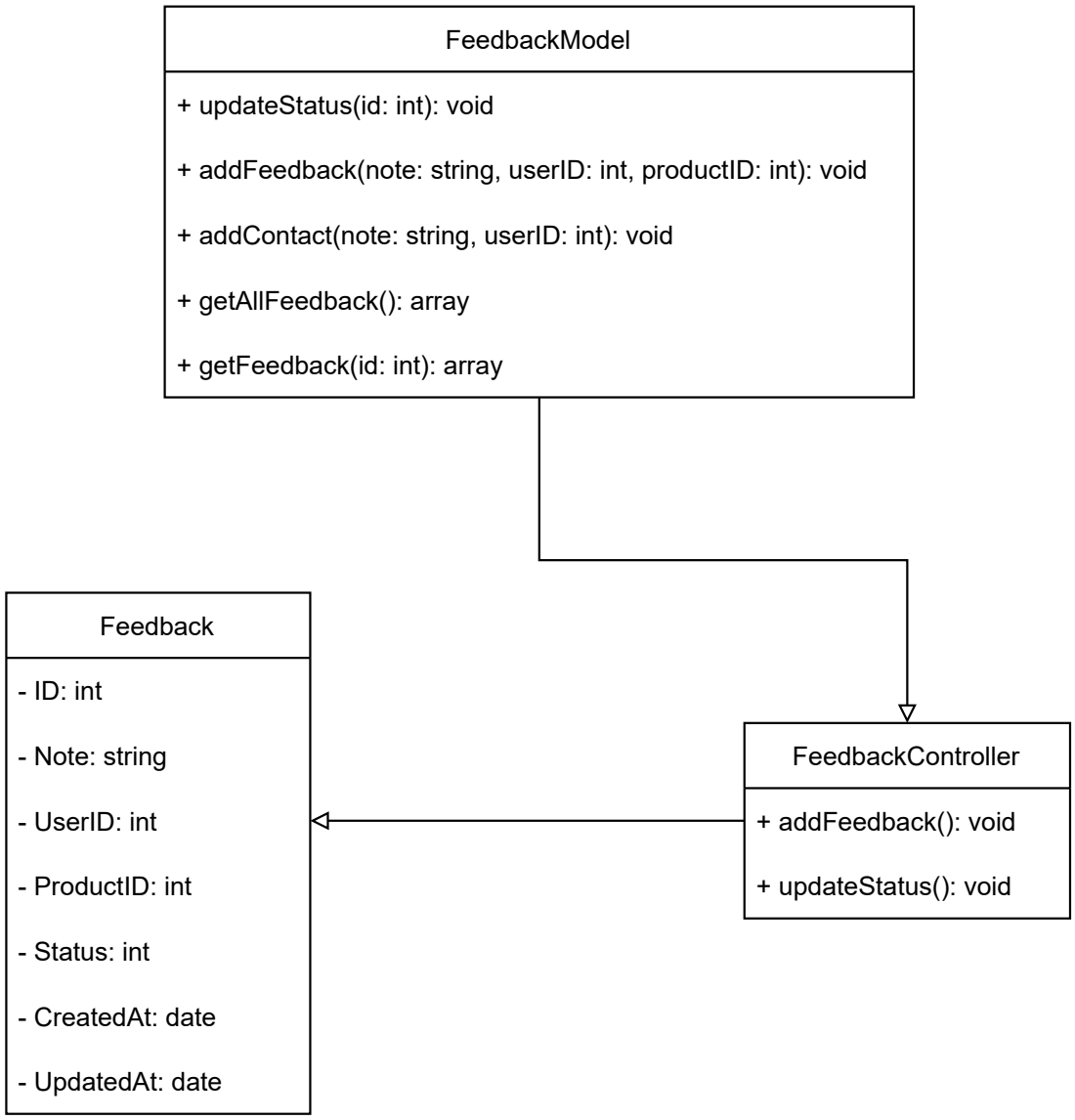


Logical Design Diagram

Additionally, we also created some class diagrams to represent the system's constituent parts, show how they are related, and explain the functions and services they offer.



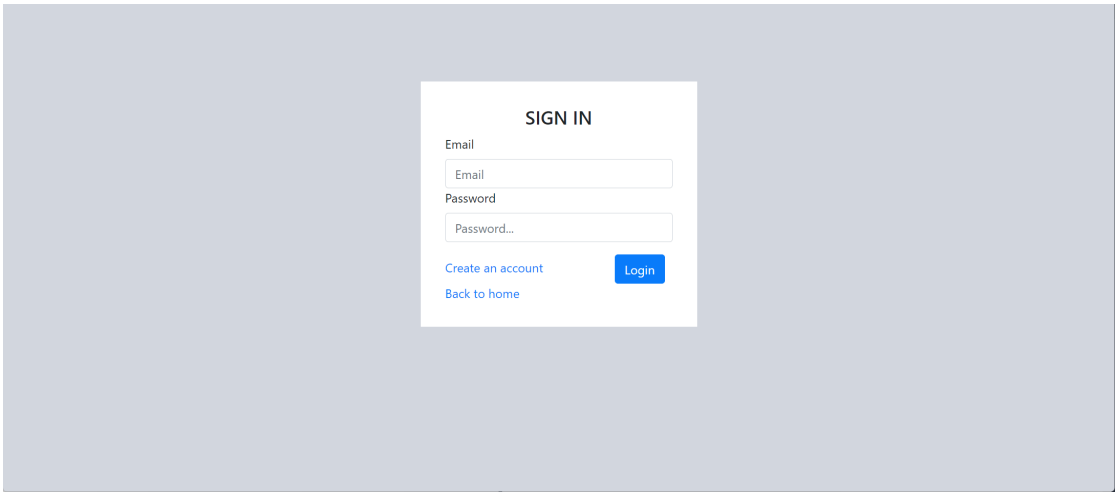
User Class Diagram



Feedback Class Diagram

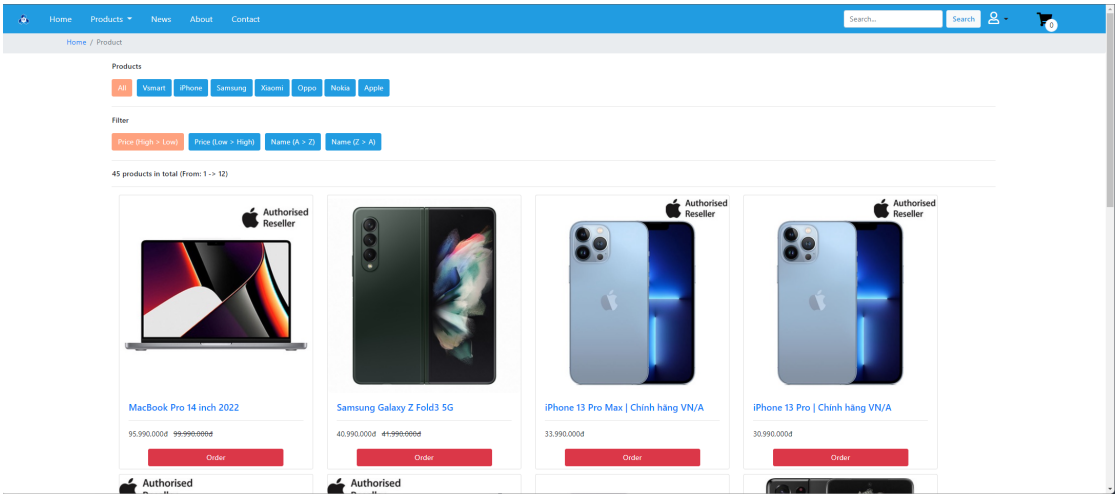
4.3 User Interface

Here are a few images of our website's user interface.

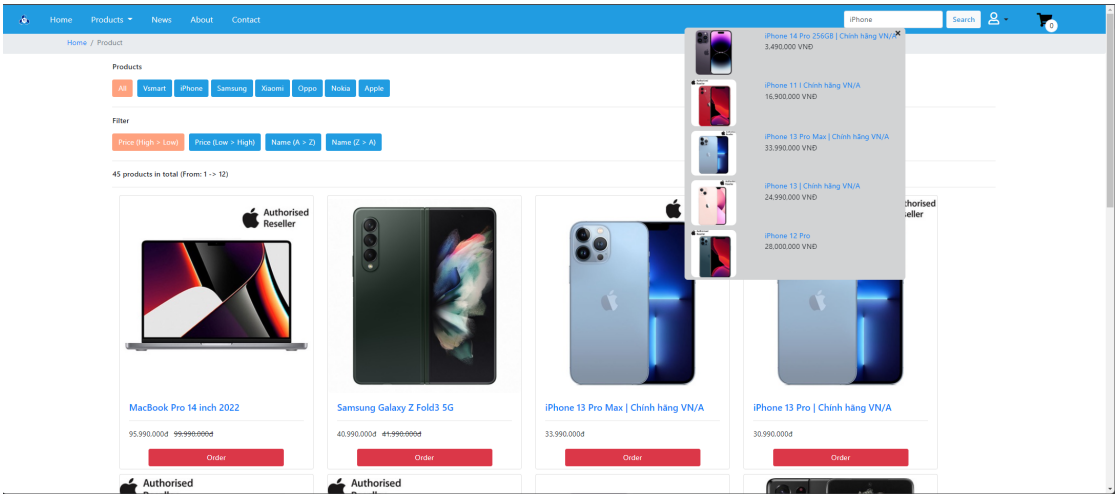


Login page

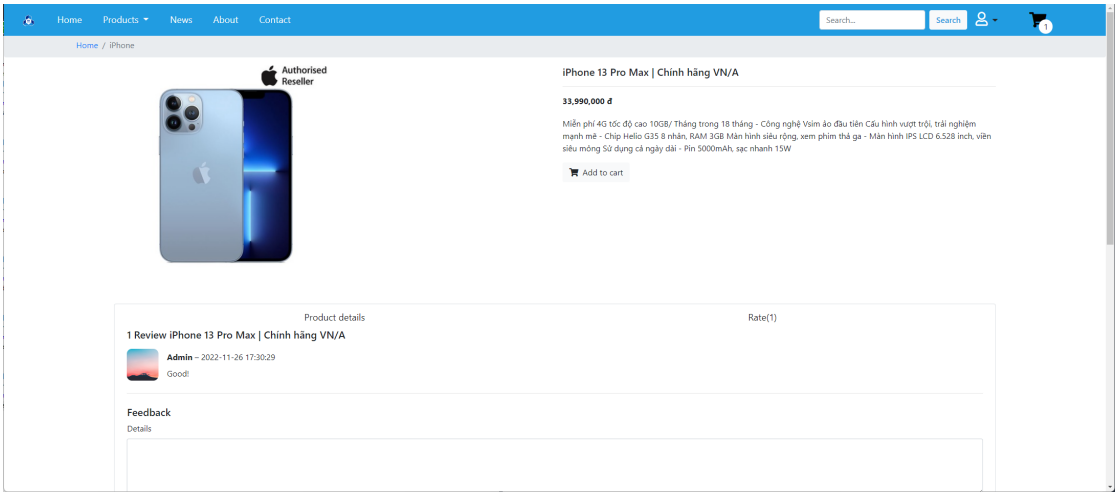
Home page



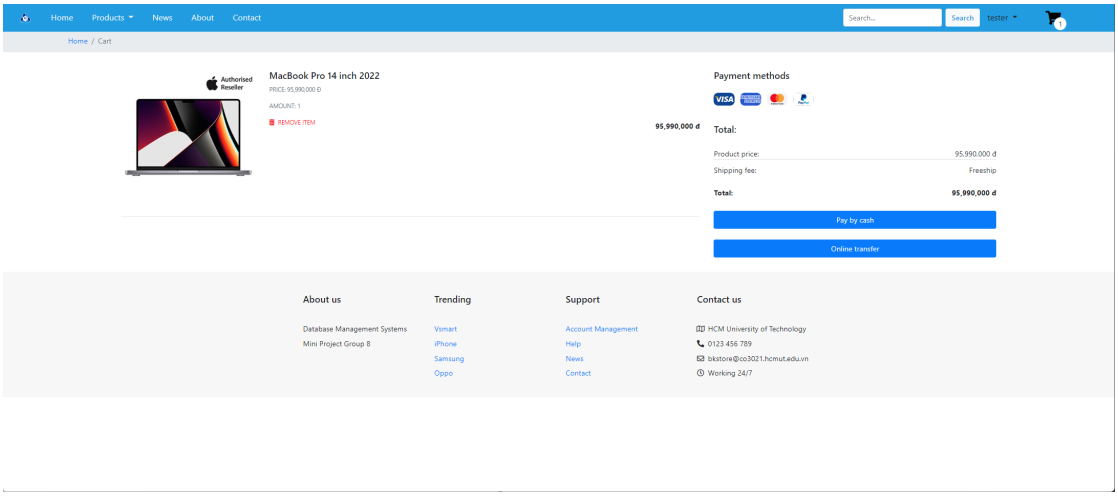
Product page with categories and filters



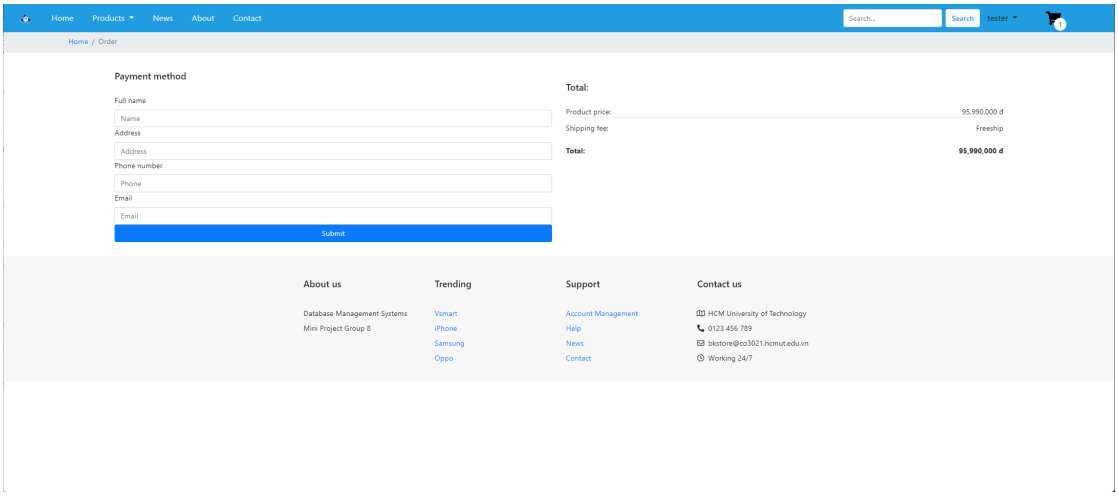
Searching box



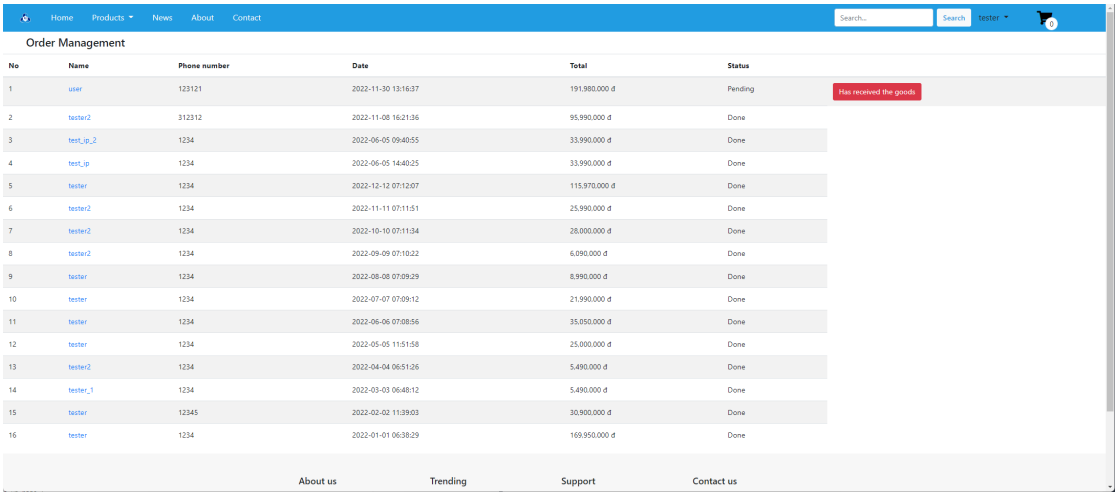
Product detail page



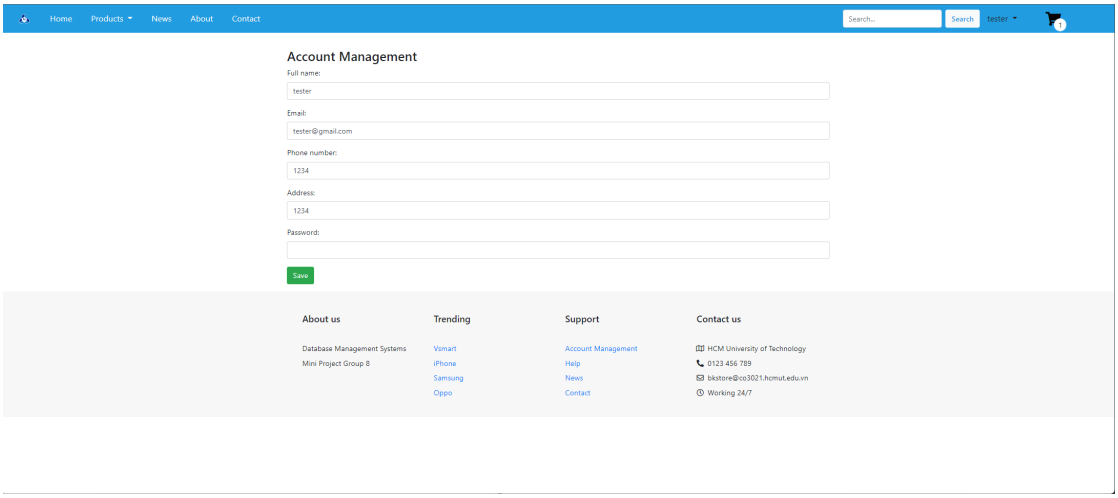
Payment cart page



Payment information page



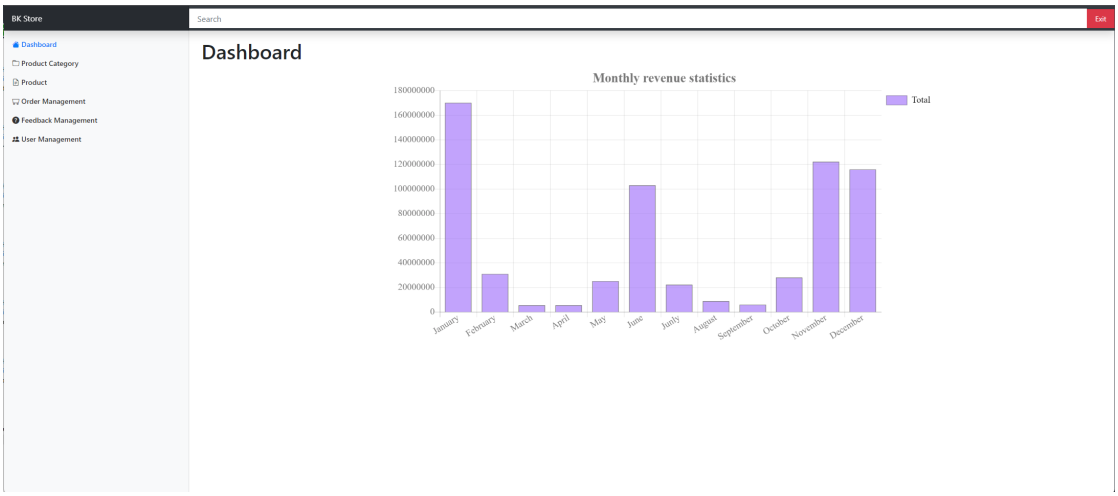
Order status page



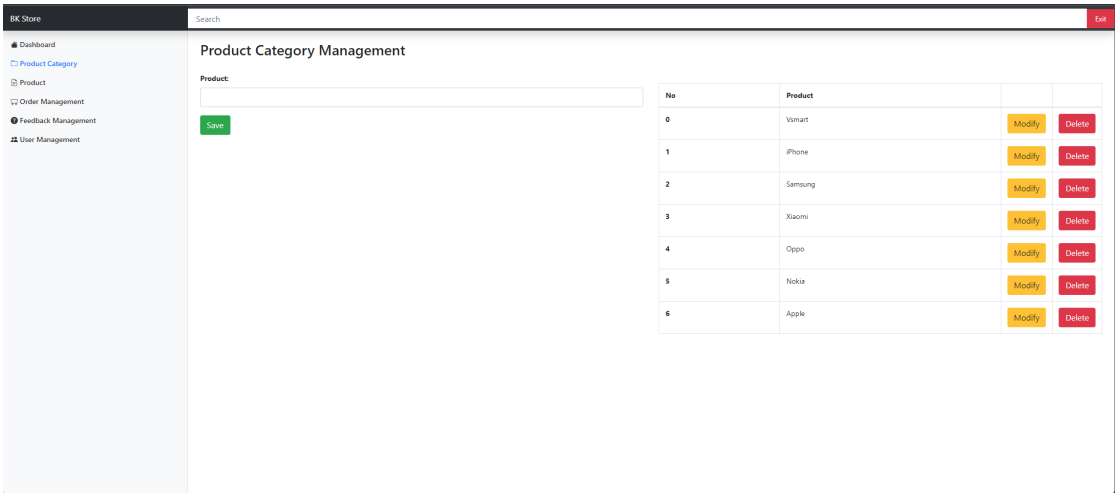
Account page

4.4 Admin Interface

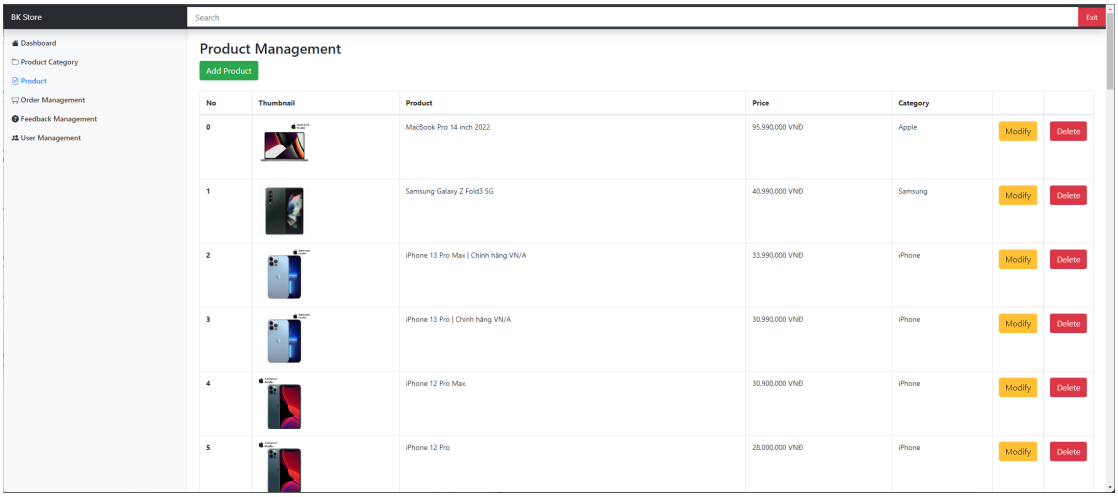
Here are a few images of our website's admin interface.



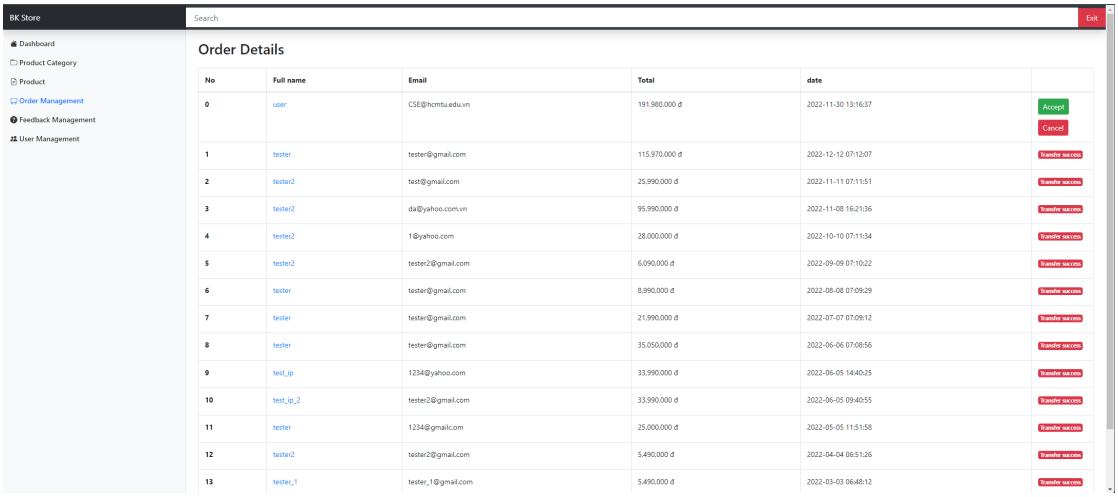
Dashboard page



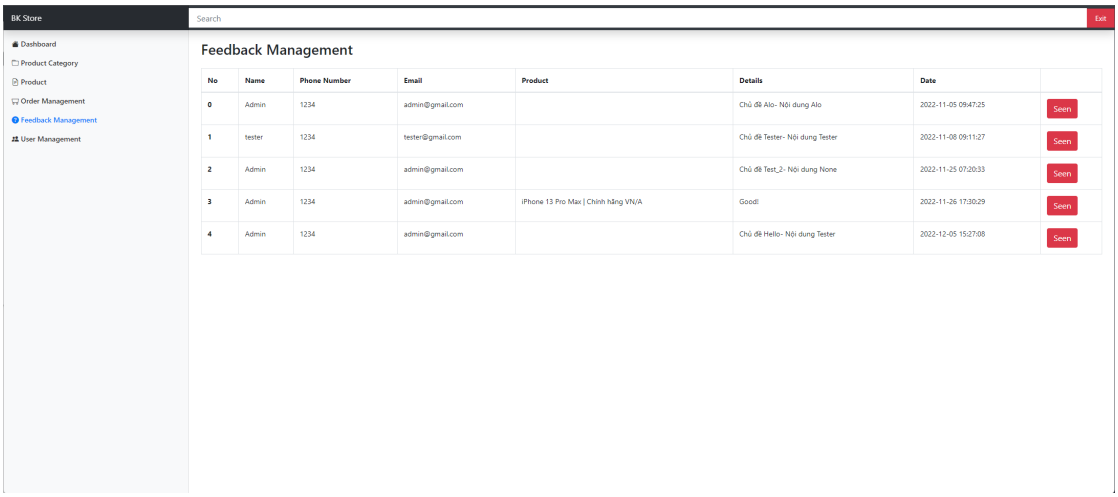
Product category management page



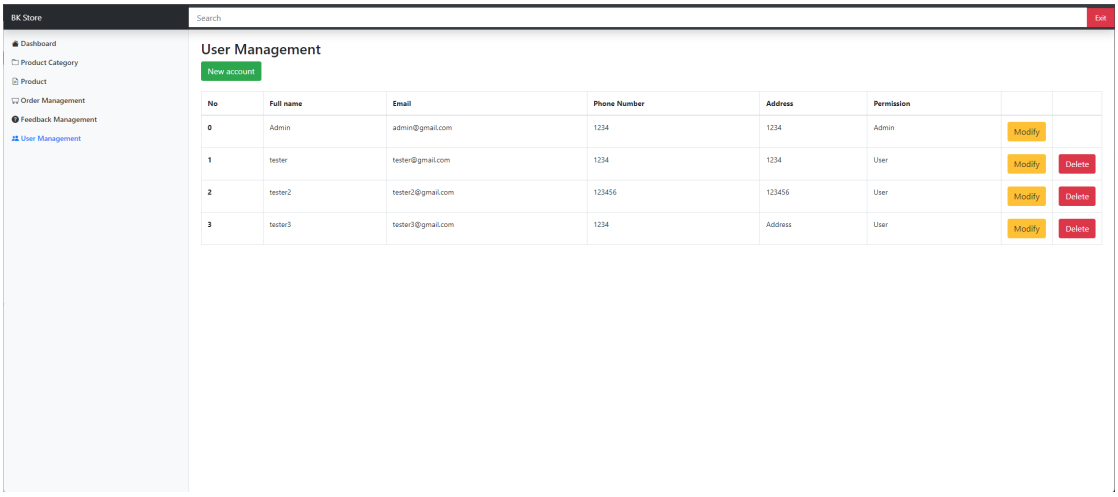
Product management page



Order management page



Feedback management page



User management page

4.5 Demonstration

For demonstration part, please watch the video `DBMS_G8_1952207_1752209_1952386_1952996_Application_Demonstration.mp4` included in our submission.

5 Conclusion

In this section, we will present our conclusions, summary and comparison according to the requirements of the two databases PostgreSQL and HBase. Note that, for the comparison to be meaningful (to some extent) we use a common dataset for both databases.

5.1 Data Storage and File Sturctures

PostgreSQL stores its data files in the same location as its configuration and control files for a database cluster, and it is usually in the directory specified by the `PGDATA` environment variable. These files are organized into a number of subdirectories.

All database objects in PostgreSQL are managed internally by their respective object identifiers (OIDs), which are assigned 4 byte integers. The OID of the database is stored in the `pg_database` system table. The OIDs of objects such as tables, indexes, and sequences in the database are stored in the `pg_class` system table.

Each row is stored on an (usually) 8 kB page along with several other rows. Each page in turn is part of a 1GB file. While processing a query, when PostgreSQL needs to fetch a row from disk, PostgreSQL will read the entire page the row is stored on. This is, at a high level, how PostgreSQL represents data stored in it on disk.

On the other hand, Hbase store on HDFS, it runs on the top of HDFS and provides all capabilities of a large table to Hadoop. It has the capabilities to store and process a billion rows of data at a time, also the storage capacity can be increased at any time due to its dynamic feature.

HBase makes use of HDFS and Zookeeper as well as other Hadoop components. HMaster and Zookeeper coordinate to manage the area server. There are many regions on a region server. The handling, managing, and reading/writing operations for the region are carried out by the region server.

A region contains information for each column and its qualifiers. A HBase table can be split up into a number of regions, with each region server managing a group of regions. HBase became famous due to its features of fast and random read/write operation.

5.2 Indexes

An index is a structure or object that makes it easier to quickly retrieve particular rows or data. Depending on the query requirements, indexes can be constructed using a single column, a group of columns, or even partial data. The supplied table's index will produce a pointer to the real records.

The index in a database is remarkably similar to an index in a back of a book. A book's index, which includes all themes alphabetically, can be consulted before referring to one or more specific page numbers if you want to reference all the pages in the book that address a particular topic.

GiST, SP-GiST, GIN, and BRIN are supported natively by PostgreSQL together with normal B-tree, Hash, and GiST. Each sort of index uses a unique algorithm that is best suited for a

certain type of query. Note that, an index helps to speed up `SELECT` queries and `WHERE` clauses. However, it slows down data input, with `UPDATE` and `INSERT` statements. Indexes can be created or dropped with no effect on the data. The `CREATE` INDEX command produces B-tree indexes by default, which are ussually a suitable choice. Indexes in PostgreSQL also support the following:

- Expression indexes
  Ceated with an index of the result of an expression or function, instead of simply the value of a column.
- Partial indexes
  Index only a part of a table.
- Multifield/Compound indexes
  Index multiple fields of a table.
- Geospatial Indexes
  Index to support geospatial queries.

All indexes in PostgreSQL are secondary indexes, meaning that each index is stored separately from the table's main data area (which is called the table's heap in PostgreSQL terminology).

Despite the fact that indexes are meant to improve a database's efficiency, there are some situations in which they ought to be avoided. When an index should no longer be used, as indicated by the following rules [21]:

- Indexes should not be used on small tables.
- Tables that have frequent, large batch update or insert operations.
- Indexes should not be used on columns that contain a high number of `NULL` values.
- Columns that are frequently manipulated should not be indexed.

Due to the **lack of records** in the database to which our application connects, it is not possible to tell the difference between a query supported by the index or not (only about 50 products, 20 transactions , several accounts and others). Therefore, we create a `mock_db` database with just one `mock_data` table and the following fields:

```
CREATE TABLE mock_data
(
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    gender VARCHAR(50),
    birthday DATE,
    email VARCHAR(50),
    state VARCHAR(50),
    company VARCHAR(50),
    job VARCHAR(50),
    salary INT,
    credit_card_type VARCHAR(50)
);
```

This is a structured table containing the information of 22222 people living in the United States, detailed as follows:

- `first_name`
  This is the field that represents the person's first name, has 7703 distinct values.
- `last_name`
  This is the field that represents the person's last name, has 17100 distinct values.
- `gender`
  This is the field that represents the person's gender which is divided in to 8 types `Genderqueer`, `Bigender`, `Genderfluid`, `Male`, `Polygender`, `Non-binary`, `Female` and `Agender`.
- `birthday`
  This is the field that represents the person's birthday, has 12870 distinct values, range from `1970-01-01` to `2022-12-16`.
- `email`
  This is the field that represents the personal email, has 21222 distinct values.
- `state`
  This is the field that represents the state in which the person lives, has 51 distinct values.
- `company`
  This is the field that represents the person's company name, has 383 distinct values.
- `job`
  This is the field that represents the person's job title, has 195 distinct values.
- `salary`
  This is the field that represents the person's salary, has 19076 distinct values, range from 0 to 100000.
- `credit_card_type`
  This is a field that represents the type of card the person uses. Because living in the US, this field has only one value, the `mastercard`.

Suppose we want to find 10 people whose salary is greater than 50000, we can use the following query:

```
SELECT first_name, last_name, company, job, salary FROM mock_data WHERE salary >
50000 LIMIT 10;
```

```
mock_db=# SELECT first_name, last_name, company, job, salary FROM mock_data WHERE
salary > 50000 LIMIT 10;
 first_name |  last_name   |  company  |             job              | salary
------------+--------------+-----------+------------------------------+--------
 Zerk       | Kiddle       | Fanoodle  | Office Assistant III         |  95468
 Llewellyn  | Ableson      | Oozz      | Operator                     |  68174
 Ody        | Nunnery      | Skivee    | Environmental Tech           |  94680
 Celle      | Naptine      | Linktype  | Automation Specialist III    |  78865
 Flynn      | Klimontovich | Blogtags  | Sales Associate              |  92681
 Haskell    | Burlay       | Nlounge   | Clinical Specialist          |  95565
 Bryon      | Stobbart     | Oba       | Computer Systems Analyst III |  82681
 Kit        | Linton       | Wikizz    | Operator                     |  82785
 Braden     | Andresen     | Fanoodle  | Cost Accountant              |  80613
 Raquela    | Hurch        | Riffpedia | Staff Accountant II          |  82041
```

```
(10 rows)
```

Actually with the last query, PostgreSQL will find from top to bottom 10 records that meet the conditions, to make it a little more difficult, we will query the person whose salary is exactly equal to a specific number, for example term is 50032:

```
SELECT first_name, last_name, company, job, salary FROM mock_data WHERE salary =
500032;
```

```
mock_db=# SELECT first_name, last_name, company, job, salary FROM mock_data WHERE
salary = 50032;
 first_name | last_name | company |      job      | salary
------------+-----------+---------+---------------+--------
 Ophelia    | Skilton   | Skiba   | Actuary       |  50032
 Shoshana   | Ruppel    | Plajo   | Senior Editor |  50032
 Ophelia    | Skilton   | Skiba   | Actuary       |  50032
(3 rows)
```

Fortunately, we have three individuals (actually only 2 people, 1 person is duplicated, but just ignore it), and we can use the keyword `EXPLAIN` to be able to examine the cost of the last query:

```
EXPLAIN (ANALYZE, FORMAT JSON) SELECT first_name, last_name, company, job, salary
FROM mock_data WHERE salary = 500032;
```

```
mock_db=# EXPLAIN (ANALYZE, FORMAT JSON) SELECT first_name, last_name, company, job,
salary FROM mock_data WHERE salary = 50032;
              QUERY PLAN
---------------------------------------
 [                                    +
   {                                  +
     "Plan": {                        +
       "Node Type": "Seq Scan",       +
       "Parallel Aware": false,       +
       "Async Capable": false,        +
       "Relation Name": "mock_data",  +
       "Alias": "mock_data",          +
       "Startup Cost": 0.00,          +
       "Total Cost": 643.78,          +
       "Plan Rows": 1,                +
       "Plan Width": 44,              +
       "Actual Startup Time": 0.045,  +
       "Actual Total Time": 2.405,    +
       "Actual Rows": 3,              +
       "Actual Loops": 1,             +
       "Filter": "(salary = 50032)",  +
       "Rows Removed by Filter": 22219+
     },                               +
     "Planning Time": 0.060,          +
     "Triggers": [                    +
     ],                               +
     "Execution Time": 2.418          +
```

```
    }                                   +
  ]
(1 row)
```

"Node Type": "Seq Scan" here means that PostgreSQL is searching linearly. We now set up a B-tree index on the salary field because indexes of this type are great for greater than, less than, greater than, less than, and equal to comparisons, especially data types numeric format:

```
CREATE INDEX salary_index ON mock_data USING BTREE (salary);
```

```
mock_db=# CREATE INDEX salary_index  ON mock_data USING BTREE (salary);
CREATE INDEX
```

```
mock_db=# \d mock_data
                        Table "public.mock_data"
      Column      |         Type          | Collation | Nullable | Default
------------------+-----------------------+-----------+----------+---------
 first_name       | character varying(50) |           |          |
 last_name        | character varying(50) |           |          |
 gender           | character varying(50) |           |          |
 birthday         | date                  |           |          |
 email            | character varying(50) |           |          |
 state            | character varying(50) |           |          |
 company          | character varying(50) |           |          |
 job              | character varying(50) |           |          |
 salary           | integer               |           |          |
 credit_card_type | character varying(50) |           |          |
Indexes:
    "salary_index" btree (salary)
```

We again query with salary equal to 50032 as above and price statistics of the query:

```
mock_db=# EXPLAIN (ANALYZE, FORMAT JSON) SELECT first_name, last_name, company, job,
salary FROM MOCK_DATA WHERE salary = 50032;
                QUERY PLAN
-----------------------------------------
 [                                      +
   {                                    +
     "Plan": {                          +
       "Node Type": "Index Scan",       +
       "Parallel Aware": false,         +
       "Async Capable": false,          +
       "Scan Direction": "Forward",     +
       "Index Name": "salary_index",    +
       "Relation Name": "mock_data",    +
       "Alias": "mock_data",            +
       "Startup Cost": 0.29,            +
       "Total Cost": 8.30,              +
       "Plan Rows": 1,                  +
       "Plan Width": 44,                +
       "Actual Startup Time": 0.058,    +
```

```
        "Actual Total Time": 0.080,       +
        "Actual Rows": 3,                 +
        "Actual Loops": 1,                +
        "Index Cond": "(salary = 50032)", +
        "Rows Removed by Index Recheck": 0+
      },                                  +
      "Planning Time": 2.155,             +
      "Triggers": [                       +
      ],                                  +
      "Execution Time": 0.160             +
    }                                     +
  ]
(1 row)
```

It can be clearly seen that PostgreSQL has now switched to index-based search - `"Node Type": "Index Scan"`. And thanks to that, the cost has been greatly reduced, from 643.78 down to 8.30, which means a reduction of up to 98.7107% compared to the original. We can also see that the total query execution time has been reduced from 2.418 milliseconds down to 0.160 milliseconds, which is 93.383% faster. Another intriguing point is that, in order to get the result without using the index, we would need to scan 22219 rows, however with a B-tree index, we require none at all.

On the other side, HBase databases do not support indexes by nature.

For HBase, apart from the default primary index built on rowkey, HBase does not have native support for secondary index on any other column(s), so any queries involving non-rowkey columns will result in a full table scan, significantly reduce performance, especially on databases that have enormous tables, which is the case that people would usually consider using a non-relational DBMS like HBase for.

Fortunately, since version 0.92, HBase introduced a feature that could help user to create and mange a secondary index more efficiently, the co-processor. There are two types of co-processor in HBase, **Observer** and **Endpoint**, off of which the former, or more specifically, the RegionObserver, which can act trigger in a traditional SQL DBMS, would be useful in implementing secondary index. This is not silver bullet, however, as user would still have to create, manage and synchronize an entire separate index table. This would add a significant amount of overhead in term of complexity, time, effort and human resources for the customer as they have to ensure the synchronization between the main table and the index one themselves. It also mean that there are no special data structure like B-tree or hash to support for a more efficient index as the index is just another normal table.

Aside from co-processor, one can cover HBase in a "SQL skin" like Apache Phoenix to use SQL to interact with HBase, and more importantly, have features such as built-in support for secondary indexes just like in a normal SQL DBMS. Phoenix is designed to provide a SQL translation layer for those who would need the advantages of a NoSQL DBMS but prefer to use SQL as their primary query language. Phoenix have two main techniques of indexing, local index, which would store the index on the same server node as the main table and is suitable for write heavy operations, and global index, which would put the index on a centralized node, separate from the original data is a good choice for systems that emphasize on reading data. It is also noteworthy that they also have two special types of index, functional and covered index, both of which having their own advantages, with the former being helpful in queries that use expressions

by creating index on function rather than column, and the latter, which combine part of the data into the index table itself, can help on saving IO cost.

For easy comparison between PostgreSQl and HBase, here is a table showing the main differences in the two DBMSs:

Database	Secondary index implementation	Index type	Special data structure for index
PostgreSQL	Native	- Expression - Partial - Multifield - Geospatial	- B-tree - Hash - GiST - SP-GiST - GIN - BRIN
HBase	- Through co-processor. - Third-party SQL translation layer like Phoenix.	- Covered (≈ partial index) with Phoenix. - Functional (≈ expression index) with Phoenix.	No support

Main differences between PostgreSQL and HBase in secondary index support and implementation

To observe and compare the performance between HBase and PostgreSQL, we intend to perform the same interactions with the databases in the same `mock_db`. Hence, inserts records row-by-row is a time consuming task and requires much labour, we export the data into a `csv_format` file. In Hbase, it has a `ImportTSV` module, which help to load the data into HBase table from a `csv_format` file [10].

```
hbase org.apache.hadoop.hbase.mapreduce.ImportTsv -Dimporttsv.separator=','
-Dimporttsv.columns=HBASE_ROW_KEY,first_name,last_name,gender,birthday,email,state,
company,job,salary,credit_card_type employees
/data/employees.csv
```

Rowkey indexing is used in HBase table by defaults, actually it is lexicographically ordered by the key. By the rowkey indexing, HBase know which region service to find each key and within that regionserver the region and the sepecific HFile.

First, we scane the table the select employees, whose salary is greater than 50000, limit display to 10 with HBase command:

```
scan 'employees', {LIMIT => 10, FILTER => "ValueFilter( >, 'binaryprefix:50000')"}
```

The result is displayed in the form the following:

```
ROW                                             COLUMN+CELL
 0                                              column=company:,
 timestamp=1671287041220, value=Fanoodle
 0                                              column=credit_card_type:,
 timestamp=1671287041220, value=mastercard
 0                                              column=email:, timestamp=1671287041220,
 value=zkiddle0@europa.eu
```

```
5  0                                                column=first_name:,
   timestamp=1671287041220, value=Zerk
6  0                                                column=gender:, timestamp=1671287041220,
   value=Male
7  0                                                column=job:, timestamp=1671287041220,
   value=Office Assistant III
8  0                                                column=last_name:,
   timestamp=1671287041220, value=Kiddle
9  0                                                column=salary:, timestamp=1671287041220,
   value=95468
10 0                                                column=state:, timestamp=1671287041220,
   value=Colorado
11 ...
12 10 row(s)
13 Took 0.0200 seconds
```

Moreover, we scan the tables to the get the employees whose salary is exactly 50032:

```
1 scan 'employees', {FILTER => "ValueFilter( =, 'binaryprefix:50032')"}
```

```
1 ROW                                              COLUMN+CELL
2  170                                             column=salary:, timestamp=1671293112487,
   value=50032
3  21392                                           column=salary:, timestamp=1671293112487,
   value=50032
4  4324                                            column=salary:, timestamp=1671293112487,
   value=50032
5 3 row(s)
6 Took 0.2385 seconds
```

It took the HBase 0.2385 seconds to scan all the HFile to find the suitable records, the execution time of HBase for this command is longer than the previous one, but it still not a considerable output.

5.3 Query Processing and Optimization

When PostgreSQL receives a query, the query is first parsed into an internal representation, which goes through a series of transformations, resulting in a query plan that is used by the executor to process the query.

The first stage of a query's transformation is the rewrite stage, which is responsible for implementing the PostgreSQL rules system. In PostgreSQL, users can create rules that are fired on different query structures such as update, delete, insert, and select statements. A view is implemented by the system by converting a view definition into a select rule. When a query involving a select statement on the view is received, the select rule for the view is fired, and the query is rewritten using the definition of the view.

A rule is registered in the system using the create rule command, at which point information on the rule is stored in the catalog. This catalog is then used during query rewrite to uncover all candidate rules for a given query.

The rewrite phase first deals with all update, delete, and insert statements by firing all appropriate rules. Such statements might be complicated and contain select clauses. Subsequently, all the remaining rules involving only select statements are fired. Since the firing of a rule may cause the query to be rewritten to a form that may require another rule to be fired, the rules are repeatedly checked on each form of the rewritten query until a fixed point is reached and no more rules need to be fired.

Once the query has been rewritten, it is subject to the planning and optimization phase. Here, each query block is treated in isolation and a plan is generated for it. This planning begins bottom-up from the rewritten query's innermost subquery, proceeding to its outermost query block.

The optimizer in PostgreSQL is, for the most part, cost based. The idea is to generate an access plan whose estimated cost is minimal. The cost model includes as parameters the I/O cost of sequential and random page fetches, as well as the CPU costs of processing heap tuples, index tuples, and simple predicates. Optimization of a query can be done using one of two approaches:

- Stadard planner
- Genetic query optimizer

The executor module is responsible for processing a query plan produced by the optimizer. The executor is based on the demand-driven pipeline model , where each operator implements the iterator interface with a set of four functions: `open()`, `next()`, `rescan()`, and `close()`). PostgreSQL iterators have an extra function, `rescan()`, which is used to reset a subplan (say for an inner loop of a join) with new values for parameters such as index keys. Some of the important categories of operators are as follows:

1. Access methods.
2. Join methods.
3. Sort.
4. Aggregation.

In PostgreSQL (unlike some commercial systems) active-database features such as triggers and constraints are not implemented in the rewrite phase. Instead they are implemented as part of the query executor. When the triggers and constraints are registered by the user, the details are associated with the catalog information for each appropriate relation and index. The executor processes an update, delete, and insert statement by repeatedly generating tuple changes for a relation. For each row modification, the executor explicitly identifies, fires, and enforces candidate triggers and constraints, before or after the change as required.

Beside that, in PostgreSQL, `EXPLAIN` is a keyword that gets prepended to a query to show a user how the query planner plans to execute the given query. Depending on the complexity of the query, it will show the join strategy, method of extracting data from tables, estimated rows involved in executing the query, and a number of other bits of useful information. Used with `ANALYZE`, `EXPLAIN` will also show the time spent on executing the query, sorts, and merges that could not be done in-memory, and more. This information is invaluable when it comes

to identifying query performance bottlenecks and opportunities, and helps us understand what information the query planner is working with as it makes its decisions for us[12].

A Cost-Based Approach

To the query planner, all the data on disk is basically the same. To determine the fastest way to reach a particular piece of data requires some estimation of the amount of time it takes to do a full table scan, a merge of two tables, and other operations to get data back to the user. PostgreSQL accomplishes this by assigning costs to each execution task, and these values are derived from the `postgresql.conf` file (see parameters ending in`*_cost` or beginning with `enable_*`). When a query is sent to the database, the query planner calculates the cumulative costs for different execution strategies and selects the most optimal plan (which may not necessarily be the one with the lowest cost).

```
test=# EXPLAIN SELECT * FROM mock_data a JOIN mock_data b ON (a.salary=b.salary)
WHERE a.salary < 50000;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=782.04..1662.40 rows=12488 width=190)
   Hash Cond: (b.salary = a.salary)
   ->  Seq Scan on mock_data b  (cost=0.00..588.56 rows=22256 width=95)
   ->  Hash  (cost=644.20..644.20 rows=11027 width=95)
         ->  Seq Scan on mock_data a  (cost=0.00..644.20 rows=11027 width=95)
               Filter: (salary < 50000)
(6 rows)
```

Here, we can see that the `Seq Scan` on `mock_data a` has cost 644.20 to execute the task. Where does this value come from? If we look at some settings and do the calculations, we find:

```
cost = ( #blocks * seq_page_cost ) + ( #records * cpu_tuple_cost ) + ( #records *
cpu_filter_cost )
test=# SELECT pg_relation_size('mock_data');
 pg_relation_size
------------------
          2998272
(1 row)
block_size      = 8192   (8kB, typical OS)
#blocks         = 366   (relation_size / block_size)
#records        = 22256
seq_page_cost   = 1      (default)
cpu_tuple_cost  = 0.01   (default)
cpu_filter_cost = 0.0025 (default)
cost = ( 366 * 1 ) + ( 22256 * 0.01 ) + ( 22256 * 0.0025 ) = 644.20
```

As we can see, the costs are directly based on some internal statistics that the query planner can work with.

When an `EXPLAIN` is prepended to a query, the query plan gets printed, but the query does not get run. We do not know whether the statistics stored in the database were correct or not, and we do not know if some operations required expensive I/O instead of fully running in memory. When used with `ANALYZE`, the query is actually run and the query plan, along with some under-

the-hood activity is printed out. Loocking at the first query above and run `EXPLAIN ANALYZE` instead of a plain `EXPLAIN`:

```
test=# EXPLAIN ANALYZE SELECT * FROM mock_data a JOIN mock_data b ON
(a.salary=b.salary) WHERE a.salary < 50000;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=782.04..1662.40 rows=12488 width=190) (actual time=6.693..13.922
 rows=14454 loops=1)
   Hash Cond: (b.salary = a.salary)
   ->  Seq Scan on mock_data b  (cost=0.00..588.56 rows=22256 width=95) (actual
   time=0.029..1.155 rows=22222 loops=1)
   ->  Hash  (cost=644.20..644.20 rows=11027 width=95) (actual time=6.551..6.552
   rows=11006 loops=1)
         Buckets: 16384  Batches: 1  Memory Usage: 1573kB
         ->  Seq Scan on mock_data a  (cost=0.00..644.20 rows=11027 width=95)
         (actual time=0.013..4.785 rows=11006 loops=1)
               Filter: (salary < 50000)
               Rows Removed by Filter: 11216
 Planning Time: 0.606 ms
 Execution Time: 14.495 ms
(10 rows)
```

There is more information – actual time and rows, as well as planning and execution times. If adding `BUFFERS`, like `EXPLAIN (ANALYZE, BUFFERS)`, cache hit/miss statistics will be displayed in the output:

```
test=# EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM mock_data a JOIN mock_data b ON
(a.salary=b.salary) WHERE a.salary < 50000;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=782.04..1662.40 rows=12488 width=190) (actual time=4.605..14.289
 rows=14454 loops=1)
   Hash Cond: (b.salary = a.salary)
   Buffers: shared hit=732
   ->  Seq Scan on mock_data b  (cost=0.00..588.56 rows=22256 width=95) (actual
   time=0.016..1.373 rows=22222 loops=1)
         Buffers: shared hit=366
   ->  Hash  (cost=644.20..644.20 rows=11027 width=95) (actual time=4.520..4.521
   rows=11006 loops=1)
         Buckets: 16384  Batches: 1  Memory Usage: 1573kB
         Buffers: shared hit=366
         ->  Seq Scan on mock_data a  (cost=0.00..644.20 rows=11027 width=95)
         (actual time=0.022..2.999 rows=11006 loops=1)
               Filter: (salary < 50000)
               Rows Removed by Filter: 11216
               Buffers: shared hit=366
 Planning Time: 0.153 ms
 Execution Time: 14.932 ms
(14 rows)
```

Since HBase has no SQL language support, the only way to tune the performance of this database is to change its configuration on the architecture. Suppose that we put the configuration of HBase

in one file named `config-hbase.sh`, this file will set the HBase configuration into default if there is no special specification. Our group has described several parameters of the HBase support for tuning performance. In this section, we choose to modify `HBASE_HEAPSIZE` to larger value (8000) than default (4000) and observe the performance on `scan` execution.

Observe the output of:

```
scan 'employees', {FILTER => "ValueFilter( =, 'binaryprefix:50032')"}
```

We get the better result in this `scan` command:

```
ROW                                                  COLUMN+CELL
 170                                                 column=salary:, timestamp=1671294387604,
 value=50032
 21392                                               column=salary:, timestamp=1671294387604,
 value=50032
 4324                                                column=salary:, timestamp=1671294387604,
 value=50032
3 row(s)
Took 0.1053 seconds
```

However, these parameters should be tuned with the dataset before changing, as different dataset will result in different best HBase configuration. Not always the larger, the better, but the admin has to choose wisely on the value of parameters configuration since there are always a trade-off when modify those values.

6 Summary and Self's Assessment

Overall, we have studied and compared all the required requirements which are Data Storage and File Structures (extra requirement), Indexes, Query Processing and Optimization for the two DBMS PostgreSQL and HBase. With Postgres being SQL and HBase being a NoSQL DBMS, they are vastly different in the way they store, process and have supporting features to manage data. We have learned a lot from carrying out this project. We have gained a great deal of knowledge on the internal structure of each DBMS, the way they handle and manage data, as well as how to install, set up, interact and use each one in a practical application.

From the comparison we have drawn, each of these two DBMS are fundamentally different and each are designed with different use cases in mind. For example, PostgreSQL, being a relational DBMS, is suitable for well-structured and well-defined schemas with native support for popular data-mange features like secondary index and using SQL for query and interacting with the DBMS. On the other hand, HBase, a non-relational DBMS, is a great choice for applications with unstructured and dynamic, ever-changing data. It is also good when you have a large amount of data and you need a way to store, retrieve and manage them with concern for high performance computing as HBase is a distributed DBMS with multiple nodes spanning across multiple servers, which would have a impact on both performance and reliability.

In summary, non of these two DBMS are considered to be better than the other. Each of them can shine in their own way, and the choice of which DBMS to use depend on your application and the nature and characteristic of the data that you are going to use.

Regarding self's assessment, this massive assignment taught us a lot. For instance, how to effectively discover and read documents, how to connect an application to two distinct databases, etc. But most significantly, we developed teamwork skills. It is true that completing this assignment will take a lot of work because it is so challenging and time-consuming. And so we took a lot of time and effort to make this report as complete as we could in order to complete the whole. To do that was only possible thanks to the cooperation of the 4 of us. Therefore, we evaluate our own assigned tasks on this crucial assignment as great, praise each team member for their tireless efforts, and state that we gave it our all. Indeed, based on what is demonstrated in this report, we believe that these are unforgettable experiences while accomplishing this assignment. Additionally, due to the huge volume in this assignment, errors are almost inevitable to occur (if any). Please disregard (if at all possible) any mistakes or get in touch with us so we can correct them.

Concerning the details of the complete progress over time, please check the `DBMS_G8_1952207_1752209_1952386_1952996_Progress_Time_Report.pdf` file attached to our submission.

In the end, we would like to express our sincere gratitude and respect to Dr. Vo Thi Ngoc Chau for all of her assistance, instruction, and support throughout this assignment as well as in the Database Management System course.

7 Workload

Member	ID	Workload
Nguyen Dinh Khuong Duy	1952207	Study material and write report on HBase secondary index approaches and implementation. Provide comparison between indexing in PostgreSQL and HBase. Provides practice examples for HBase. Write Summary and Self's Assessment. Define the functional requirements for the application. Develop user interface for the application.
Le Huy Hoang	1752209	Study material for File Structure and Data Strorage, Indexes and Query Optimization of PostgreSQL. Conclusion for File Structure and Data Strorage, Query Processing and Optimization for PostgreSQL. Provides practice examples for PostgreSQL. Generate datasets. Define the non-functional requirements for the application. Develop backend for the application.
Tran Tien Phat	1952386	Plan workloads for each team member. Write Introduction. Write Summary and Self's Assessment. Research on Indexes and Query Processing in PostgreSQL. Conclusion for Indexes. Provides practice examples for PostgreSQL. Prepare and format the reports. Generate datasets. Design and develop the application architecture. Make the demonstration for the application.
Nguyen Vo Hoang Thi	1952996	Study material for File Structure and Data Strorage, Indexes and Query Optimization of HBase. Conclusion for File Structure and Data Strorage, Query Processing and Optimization for HBase. Record progress during assigned working time. Provides practice examples for HBase. Develop admin interface for the application. Develop backend for the application.

Workload Table

8 References

References

[1] *Art of Phoenix Secondary Indexes*. URL: https://community.cloudera.com/t5/Community-Articles/Art-of-Phoenix-Secondary-Indexes/ta-p/247815 (visited on 12/10/2022).

[2] *BK Store*. URL: https://github.com/CSEK19/hcmut-co3021 (visited on 12/19/2022).

[3] *Coprocessor Introduction*. URL: https://blogs.apache.org/hbase/entry/coprocessor_introduction (visited on 12/12/2022).

[4] N. Dimiduk and A. Khurana. *HBase in Action*. 1st ed. Manning, 2012. ISBN: 9781617290527.

[5] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. 6th ed. USA: Addison-Wesley Publishing Company, 2010. ISBN: 0136086209.

[6] Lars George. *HBase: The Definitive Guide*. 1st ed. O'Reilly Media, 2011. ISBN: 1449396100.

[7] *HBase Documentation*. URL: https://hbase.apache.org/book.html (visited on 12/10/2022).

[8] *HBase Phoenix secondary index*. URL: https://www.programmerall.com/article/36962004398/ (visited on 12/10/2022).

[9] *HBase tuning performance*. URL: https://www.ibm.com/docs/en/db2-big-sql/5.0.3?topic=performance-general-hbase-tuning (visited on 12/12/2022).

[10] *HBase tutorials*. URL: https://www.tutorialspoint.com/hbase/ (visited on 12/03/2022).

[11] *HBase's coprocessor*. URL: https://programmerall.com/article/21401301474/ (visited on 12/12/2022).

[12] *How to Use EXPLAIN ANALYZE for Planning and Optimizing Query Performance in PostgreSQL*. URL: https://www.enterprisedb.com/blog/postgresql-query-optimization-performance-tuning-with-explain-analyze (visited on 11/18/2022).

[13] *Indexes in PostgreSQL*. URL: https://postgrespro.com/blog/pgsql/3994098 (visited on 12/10/2022).

[14] *Indexes in PostgreSQL-2*. URL: https://postgrespro.com/blog/pgsql/4161264 (visited on 12/10/2022).

[15] *Indexes in PostgreSQL-3*. URL: https://postgrespro.com/blog/pgsql/4161321 (visited on 12/11/2022).

[16] *Indexes in PostgreSQL-4*. URL: https://postgrespro.com/blog/pgsql/4161516 (visited on 12/12/2022).

[17] *Indexes in PostgreSQL-5*. URL: https://postgrespro.com/blog/pgsql/4175817 (visited on 12/15/2022).

[18] *Indexes in PostgreSQL-6*. URL: https://postgrespro.com/blog/pgsql/4220639 (visited on 12/15/2022).

[19] *Indexes in PostgreSQL-7*. URL: https://postgrespro.com/blog/pgsql/4261647 (visited on 12/15/2022).

[20] H.F. Korth, S. Sudarshan, and P. Abraham Silberschatz. *Database System Concepts*. 6th ed. McGraw-Hill Education, 2010. ISBN: 9780073523323.

[21] *PostgreSQL - INDEXES*. URL: https://www.tutorialspoint.com/postgresql/postgresql_indexes.htm (visited on 12/18/2022).

[22] *PostgreSQL 14.6 Documentation*. URL: https://www.postgresql.org/docs/14/index.html (visited on 12/18/2022).

[23] *PostgreSQL Query Execution Stages*. URL: https://postgrespro.com/blog/pgsql/5969262 (visited on 11/20/2022).

[24] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. 3rd ed. McGraw-Hill higher education. McGraw-Hill Education, 2003. ISBN: 9780072465631.

[25] *Secondary Indexing*. URL: https://phoenix.apache.org/secondary_indexing.html (visited on 12/10/2022).

[26] J.M. Spaggiari and K. O'Dell. *Architecting HBase Applications: A Guidebook for Successful Development and Design*. 1st ed. O'Reilly Media, 2016. ISBN: 9781491915813.

[27] *WAL in PostgreSQL: 1. Buffer Cache*. URL: https://postgrespro.com/blog/pgsql/5967951 (visited on 12/15/2022).