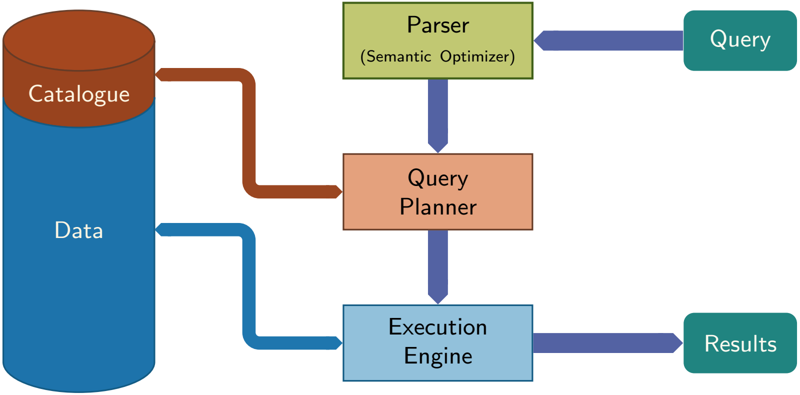
Databases

Basics of Indexation and Query Optimization

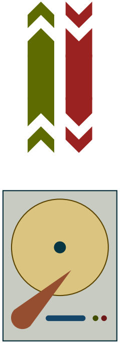
Sławek Staworko

University of Lille

Database Architecture



Memory Hierarchy



Main memory

- fast (∼20GB/s)
- expensive (∼$3/GB)
- volatile

1. Data is stored in secondary memory because of persistence considerations
2. Main performance bottleneck are data transfers between main memory and secondary memory
3. Complexity of database operations is measured in I/O operations

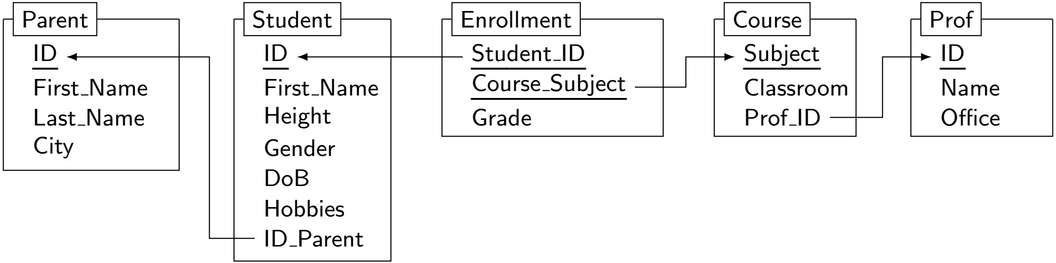
Secondary memory

- slow (∼0.1GB/s HDD; ∼0.5GB/s SSD)
- cheap (∼$0.3/GB HDD; ∼$1/GB SSD)
- persistent

Physical Data Organization

1. ## Database is a collection of files
   - One file per table
   - Files used to store the catalog with schema and statistical information
   - Files used for auxiliary structures like indexes and logs
2. ## File is a collection of blocks
   - Block is a unit of I/O access
   - Block size is a power of 2, between $2^9 = 512$B and $2^{12} = 4$KB
   - Block size is the same for the whole database
3. ## Block is a collection of records
   - Record contains data of a single table row
   - Block contains records of the same type (the same table)
   - Record may contain additional housekeeping data

Working Example: Schema

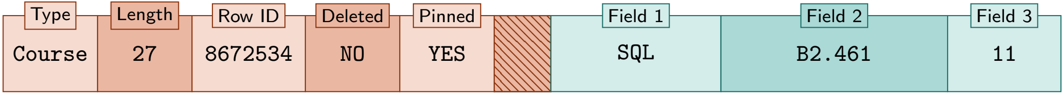



```sql
CREATE TABLE Parent (
  ID INT PRIMARY KEY,
  First_Name TEXT,
  Last_Name TEXT,
  City TEXT
);

CREATE TABLE Prof (
  ID INT PRIMARY KEY,
  Name TEXT,
  Office TEXT
);
```

```sql
CREATE TABLE Course (
  Subject TEXT PRIMARY KEY,
  Classroom TEXT,
  Prof_ID INT REFERENCES Prof(ID)
);

CREATE TABLE Enrollement (
  Student_ID INT
    REFERENCES Student(ID),
  Course_Subject TEXT
    REFERENCES Course(Subject),
  Grade FLOAT,
  PRIMARY KEY (Student_ID, Course_Subject)
);
```

```sql
CREATE TABLE Student (
  ID INT PRIMARY KEY,
  First_Name TEXT,
  Height INT,
  Gender TEXT,
  Hobbies TEXT,
  DoB DATE,
  Parent_ID INT
    REFERENCES Parent(ID)
);
```

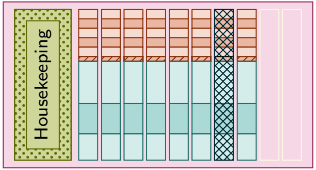
Records



Record

- a continuous chunk of memory
- has a type (e.g., table name)
- meta-data (e.g., length)
- uniquely identified (known as row ID or object ID)
- various housekeeping information:
  - **Deleted** deleted records are not erased until a scheduled or manual clean up (`VACUUM`)
  - **Pinned** if there is a pointer to the record, it must not be moved (no dangling pointers)

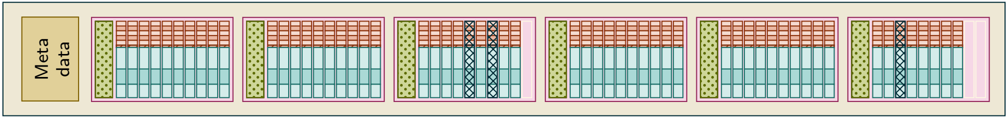
Blocks



Block

- unit of I/O access for moving data between main and secondary storage
- contains a collection of records of the same type
- may contain directory especially when storing variable-length records
- additional housekeeping information (pinned, etc.)
- block size is fixed globally: a power of 2, typically between 512B (2^9) and 4KB (2^{12})

Files



File

- an abstract data structure
- a collection of records of the same type
- stored as a set of blocks (but may be materialized on the fly)
- may contain index structures to facilitate efficient access

Elementary operations

Access `FindRecord(key)` – finds the record(s) of a given key value

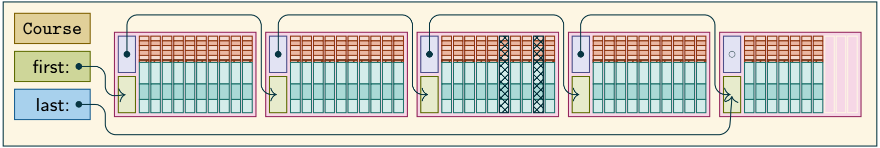
Manipulate `InsertRecord`, `DeleteRecord`, and `UpdateRecord`

Iterate `BlockIterator` – returns an iterator over all blocks used to store the file.

Iterator

- an object allowing access to all file's blocks
- two method `getNextBlock` and `hasNextBlock`

Heap Files

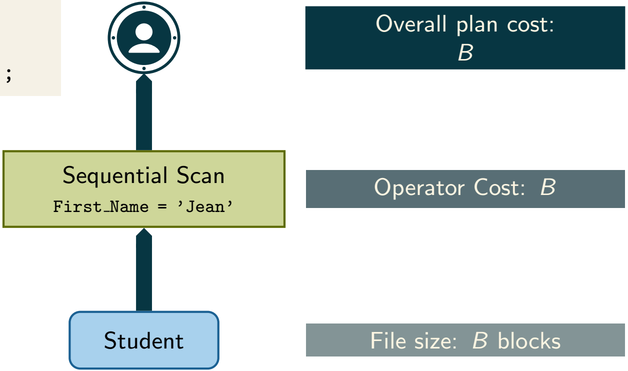


Heap file

- the simplest organization: a list of B blocks storing an unordered collection of records
- sequential search only: `FindRecord` requires B reads

Lookup query with Sequential Scan

```
SELECT *
  FROM Student
 WHERE First_Name = 'Jean';
```



Indexed Files

Index

- Structure allowing efficient lookups of records (or blocks containing relevant records)
- Defined with the index key i.e., the attribute(s) used for lookups
- May be part of the data file or stored in a separate file

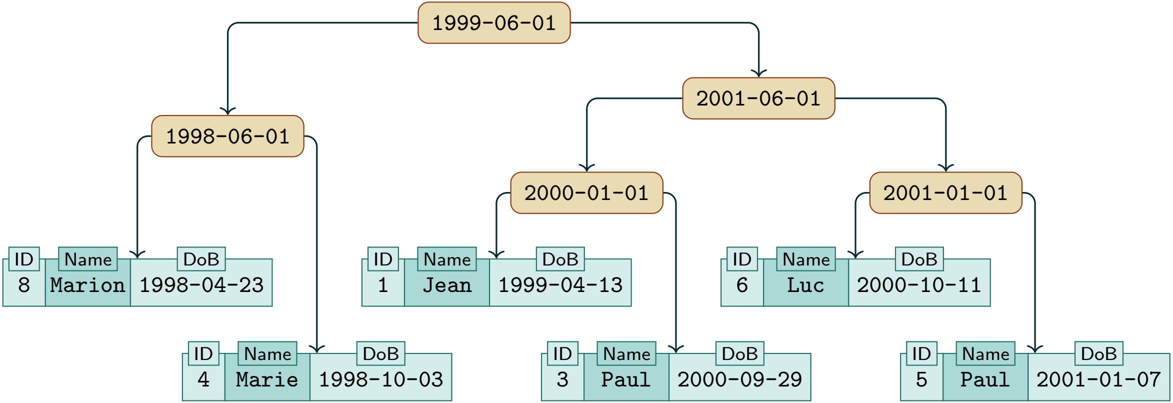
Clustered vs Unclustered

- Data file may have multiple indexes
- The data in a file may be clustered according to one selected index
- All other indexes are called unclustered

SQL

- Automatically created for primary and secondary keys (`PRIMARY KEY`, `UNIQUE`)
- `CREATE INDEX Index1 ON Student(Height);`
- PostgreSQL uses B+-tree index as default (SQLite supports only B+-tree index)
- `CREATE INDEX Index2 ON Prof USING hash(Office);`

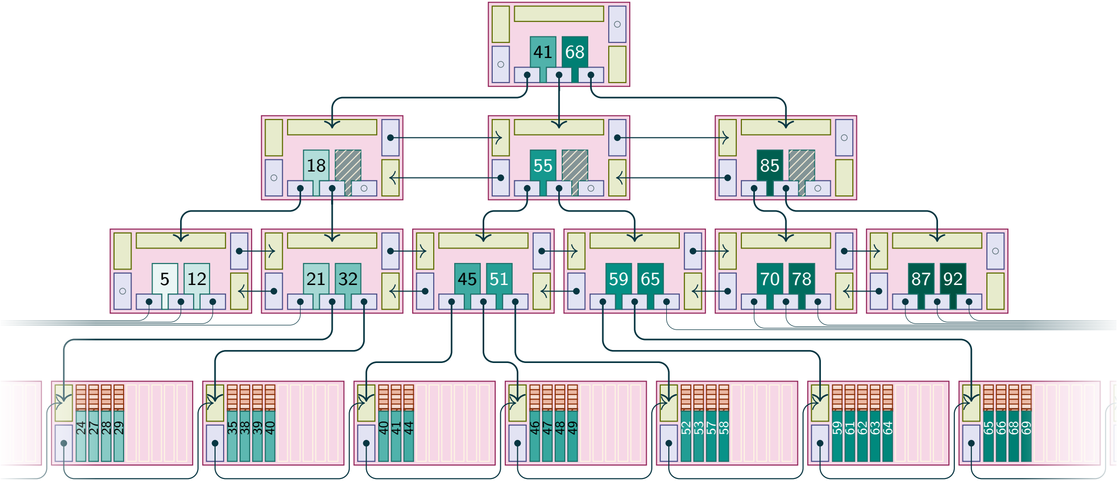
Binary Search Trees



Balanced BST

- Care is exercised to ensure the lengths of the root-to-leaf paths are uniform
- Element lookup requires $O(\log n)$ time

B+-trees

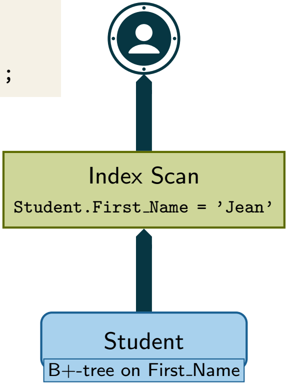


B+-Tree is a generalization of balanced binary search trees

- Node is stored in a single block and can have up to K children (typically $K \sim 1000$)
- Lookup requires time $O(\log_K n)$

Lookup query with Index Scan

```
SELECT *
  FROM Student
 WHERE First_Name = 'Jean';
```



Overall plan cost: $\log_K(B)$

Operator
Cost: $\log_K(B)$

File size: B blocks

Experiment 1: Lookup query

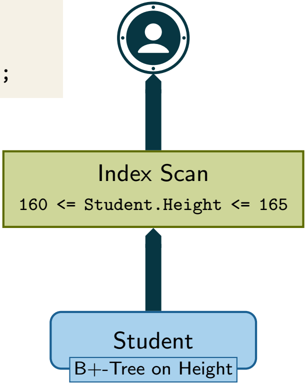
```
SELECT *
  FROM Student
 WHERE First_Name = 'SF10000';
```

```
CREATE INDEX my_index ON Student(First_Name);
```

	uni-1.db	uni-2.db	uni-3.db
1. Student line count			
2. Query run time			
3. Indexing time			
4. Query run time			

Range queries with Index Scan

```
SELECT COUNT(*)
  FROM Student
 WHERE Height BETWEEN 160 AND 165;
```



Overall plan cost: $0.16 * B * \log(B)$

Operator Cost: $0.16 * B * \log(B)$

Selectivity ratio: 16%

File size: B blocks

Experiment 2: Range query

```sql
SELECT COUNT(*)
  FROM Student
 WHERE Height BETWEEN 160 AND 165;
```

	uni-1.db	uni-2.db	uni-3.db
1. Student line count			
2. Query run time			
3. Selectivity ratio			
4. Indexing time			
5. Query run time			

Nested Loop Joins (with Scans)

```
SELECT Student.First_Name, Parent.Last_Name
  FROM Student
  JOIN Parent ON (Student.Parent_ID = Parent.ID)
 WHERE Student.DoB = '1999/04/02';
```

Nested Loop Join

```
for s in SCAN(Student, Dob='1997/04/02')
  for p in SCAN(Parent, ID=s.Parent_ID)
    output (s.First_Name,p.Last_Name)
```

Estimating the execution cost

Relevant variables:

- What is the cost of executing each scan? . . . and how many times is each scan executed?
- How many tuples is each scan likely to return?

Experiment 3: Join queries

```
SELECT Student.First_Name, Parent.Last_Name
  FROM Student
  JOIN Parent ON (Student.Parent_ID = Parent.ID)
 WHERE Student.DoB = '1999/04/02';
```

	uni-1.db	uni-2.db	uni-3.db
1. Student line count			
2. Students with DoB = '1999/04/02'			
3. Parent line count			
4. Query run time			
5. INDEX Parent(ID)			
6. Query run time			
7. INDEX Student(DoB)			
8. Query run time			

Exercise 1

Analyze and optimize the following query

```
SELECT DISTINCT Student.ID, Student.First_Name
  FROM Student
  JOIN Enrollment ON (Student.ID = Enrollment.Student_ID)
  JOIN Course ON (Enrollment.Course_Subject = Course.Subject)
  JOIN Prof ON (Course.Prof_ID = Prof.ID)
 WHERE Prof.Office = 'Office-42';
```

Exercise 2

For the following query

```
SELECT Student.First_Name, Parent.Last_Name
  FROM Student
  JOIN Parent ON (Student.Parent_ID = Parent.ID)
 WHERE Student.DoB = '1999/04/02'
   AND Parent.City = 'Lille';
```

Analyze and test independently the following two optimization strategies

- `INDEX Student(DoB)` and `INDEX Parent(ID)`
- `INDEX Parent(City)` and `INDEX Student(Parent_ID)`

Which one is more efficient and why?