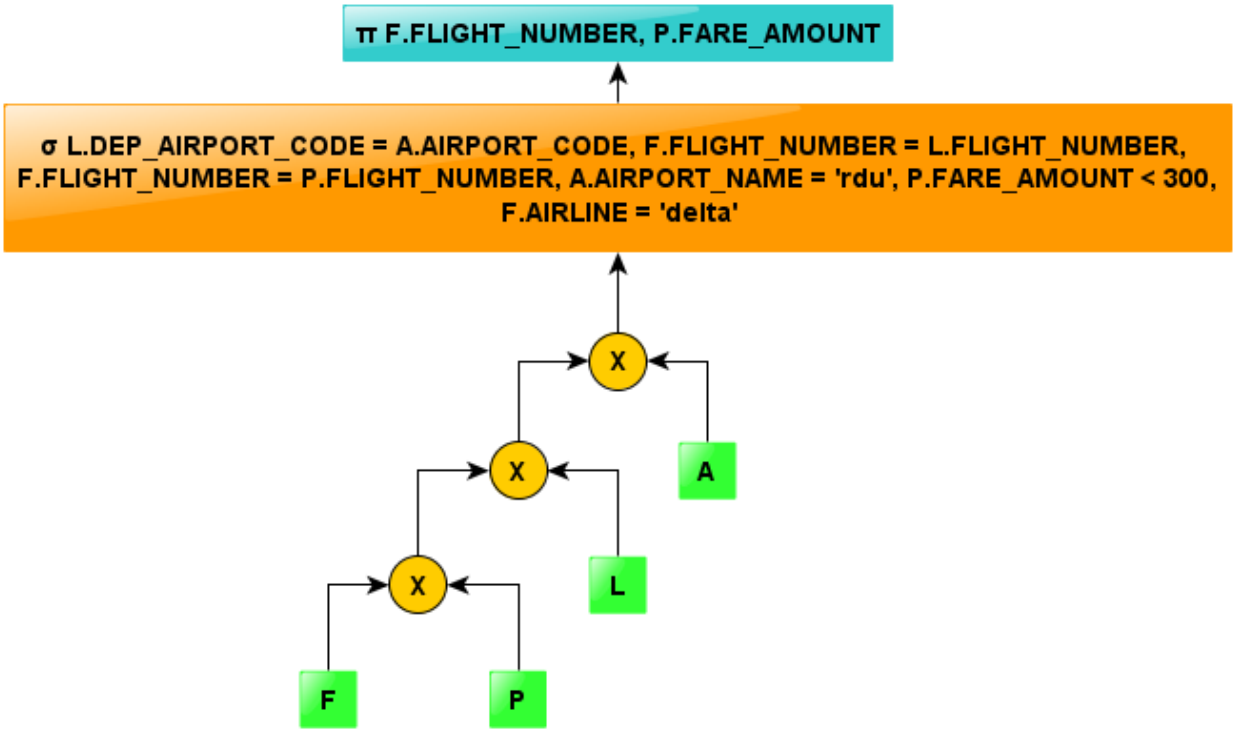
Christopher Ashley Cline
INLS 723: Databases III
Arcot Rajasekar
2012-12-07

# Final Exam

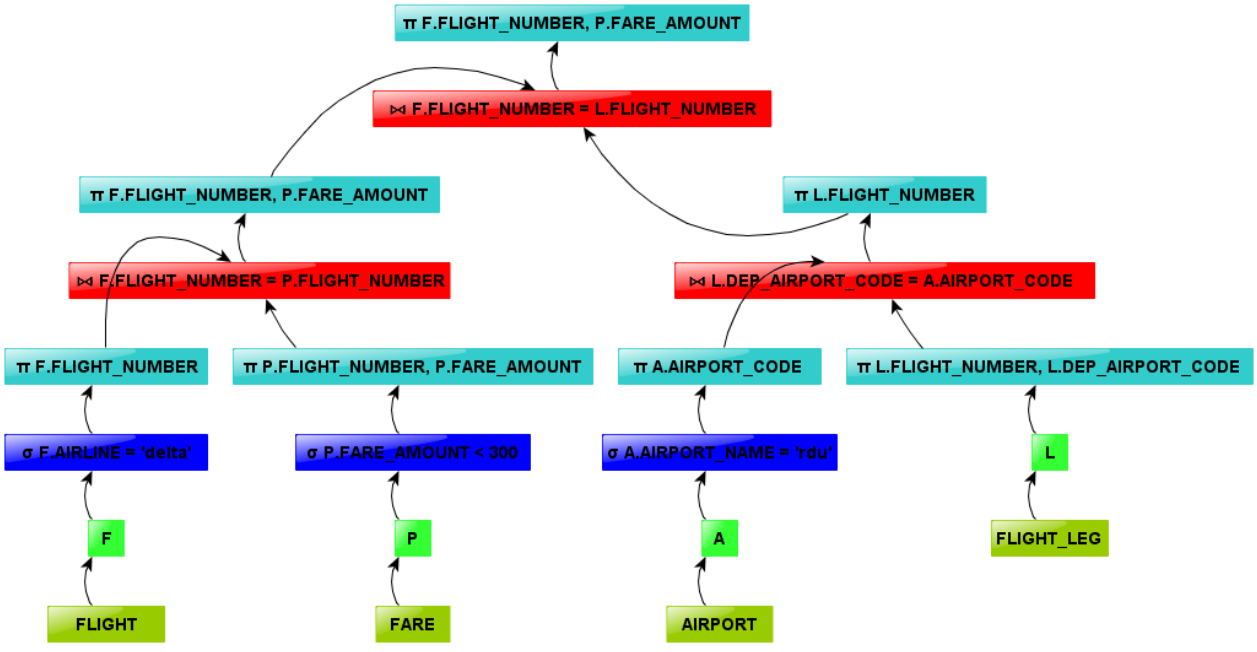
## 1. Query Optimization

### a) Simplest Query Tree:




### b) Most Cost-Effective Query Tree

- **Query Tree:**



- **Assumptions:** Several of the selection operations assume that the database tables contain additional data columns not listed in the query, and thus the projection steps are reducing the number of data columns carrying over from one operation to the next. This query tree assumes that the number of flight legs leaving RDU is fewer than the number of legs of flights offered by Delta for less than $300. Therefore the join operation of table A to table L occurs before the join operation of table F/P to table L. This could only be substantiated by analyzing statistics on the database tables. This choice also facilitates pushing the projection operation on L.FLIGHT_NUMBER further down the query tree.

**2. Query Cost**

**a)**

- **Function:** C = x + 1
- **Assumptions:** Assuming that primary index records are 1/2 the size of data file records, Table A's primary index will have twice the blocking factor of its data table, i.e. There will be 40 index records per block. Since Table A has 100 data records, its primary index will have 3 blocks. In the function above, *x* is equal to the number of levels in a multilevel index. Given an index consisting of three blocks, it will have two levels ($\log_{blfr}3 + 1$), resulting in a total cost of three block accesses.

**b)**

- **Function:** C = 2
- **Assumptions:** This answer assumes that the records from Table A where Q = 'alpha' are not spread over multiple data disc blocks. This calculated cost assumes a single block access to retrieve the hash directory, and a second to retrieve the appropriate data block.

**c)**

- **Function:** $C = x + (b_{i1}/2) + (r/2)$

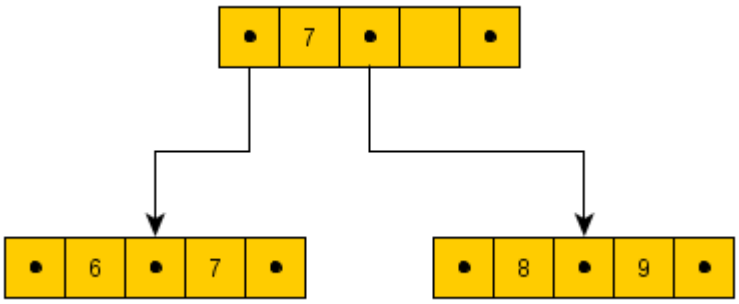
  $x$ = Number of index levels ≈ 4

  $b_{i1}$ = Number of index blocks at the lowest level ≈ 1000

  $r$ = Number of Table records ≈ 5000
- **Assumptions:** This answer assumes that approximately an equal number of records in Table B have values for column R which are less than or greater than 3000. Further, it assumes that a B+ tree index record will be 1/2 the size of a data record from Table B, resulting in twice the blocking factor, i.e. 10 records per block. Given 10,000 data rows, the B+ index should have 4 levels.

**General Assumptions:** Cost is evaluated according to access to secondary disc storage, with other query costs assumed to be minimal by comparison. In all answers, *C* is understood to be the number of disc access operations.

**3. B+ Trees**



## 4. Indexing

**a)**

- **Sorted Files:** A sorted file can be used to store the data from a database relation in an ordered fashion on disc. The file is sorted based on a selected data column. This can be helpful when performing queries based on the ordering field by allowing binary search instead of linear search. A sorted file also facilitates answering range requests dependent upon the ordering field. A file can only be sorted by a single field.
  **Example:** Given a table of employees stored as a sorted file of $b$ blocks based on Social Security Number, if a query is seeking an employee with a particular social security number, the information can be found with $\log_2 b$ block accesses instead of $b/2$ block accesses.
- **Hashing:** Hashing can be used inside a program for retrieving files based upon the value of a single field within memory, and it can also be used to retrieve external files stored on disc. Hashing is most helpful when only a single, specific record is to be returned, as opposed to range queries.
  **Example:** While finding an employee with a given SSN (a key value) is faster using a sorted file than an unsorted file, as in the example above, using a hash is even more efficient. By performing an algorithmic operation on SSN, a disc block pointer is returned, allowing the query to be answered after returning only a single data block.
- **Extendible Hashing:** Extendible hashing can be used in external storage situations. It relies upon a separately stored hash directory table. In essence, it represents multi-level hashing. Extendible hashing is helpful in circumstances where a relation is likely to grow, as extendible hashing allows for the addition of more storage buckets, while static hashing has the potential issues of unused bucket space or hashing collisons.
  **Example:** Extendible hashing could be used for the same query listed above, but would be most helpful given a context where the Employee table was constantly growing.

**b)**

- **Primary Dense Index:** Primary dense indices are most useful in circumstances where query speed is of higher priority than update efficiency, memory usage, or disc space. Having an index row for every search key value, primary indices take up more space than sparse indices. Their query speed benefit is amplified in contexts where the data table is very large.
- **Primary Sparse Index:** A sparse primary index is best used when the entire index could be contained in memory, or in contexts where a table is highly volatile, as updating a sparse index to take into account table data changes is less costly than doing the same with a dense index.
- **Clustering Index:** A clustering index is dependent upon the data file being ordered upon the relevant clustering field. A clustering index facilitates binary search and can be helpful for range queries against non-key fields.
- **Secondary Dense Index:** A secondary index is useful in contexts where the data file is already ordered on another field, but improved query efficiency is required for a given column which is a candidate key for the table.
- **Secondary Sparse Index:** A secondary sparse index is used to facilitate faster queries on non-key columns with repeating field values. It makes use of a hierarchical index structure in order to list data pointers for records which have the same value for the indexed column but exist in different locations on disc.

**c)**

- **Suggested Indices:**
  - Extendible Hash Index on F.FLIGHT_NUMBER
  - Secondary Sparse Index on F.AIRLINE

- Secondary Dense Index on A.AIRPORT_NAME
- Extendible Hash Index on L.FLIGHT_NUMBER
- Secondary Sparse Index on L.DEP_AIRPORT_CODE
- Clustering Index on P.FARE
- Secondary Dense Index on P.FLIGHT_NUMBER

- **Example Query for Multi-Column Index:** An obvious example would be a query seeking a particular combination of departure and arrival airports from the FLIGHT_LEG table. "I need a flight from RDU to ATL." : "SELECT [...] FROM FLIGHT_LEG L, [...] WHERE L.DEP_AIRPORT_CODE = 'rdu' AND L.ARR_AIRPORT_CODE = 'atl' [...];" That would be a very common query to this sort of schema, and it involves the largest table.
- **Assumptions:** I assume that A.AIRPORT_NAME is a candidate key for the AIRPORT table. I also assume that the FLIGHT and FLIGHT_LEG tables are highly volatile, and therefore both static hashing and indexing reliant upon strict file ordering are costly options. I additionally assume that the FLIGHT_LEG table has a ARR_AIRPORT_CODE as part of its schema.

**5. Disc Access**

**a)**

- **Record Size:** 300
- **Blocking Factor:** 3
- **Number of Blocks Used:** 60,000
- **Space Wasted in Each Block:** 124

**b) Total Required:** 60,000 Blocks

**c) Total Required:** The size of a secondary index with one level of indirection would depend upon the number of unique values existing in the data for the given column. As block pointers are significantly smaller than either of the table's data fields, this sort of index would represent a vast improvement over a dense index.

**6. Two-Phase Locking**

| **Transaction 1** | **Transaction 2** | **Transaction 3** |
|---|---|---|
| Read-Lock B<br>Read B<br>Write-Lock B<br>Write B<br>Read-Lock C<br>Read C<br>Write-Lock C<br>Unlock B<br>Write C<br>Unlock C | Read-Lock C<br>Read C<br>Read-Lock A<br>Read A<br>Write-Lock C<br>Write C<br>Write-Lock A<br>Unlock C<br>Write A<br>Unlock A | Read-Lock A<br>Read A<br>Read-Lock C<br>Read C<br>Write-Lock C<br>Write C<br>Write-Lock B<br>Unlock A<br>Unlock C<br>Write B<br>Write B<br>Unlock B |

**7. Transaction Scheduling**

**a) Schedule S:**

| T1 | T2 | T3 |
|---|---|---|
| | RC | |
| | | RB |
| | RB | |
| RC | | |
| RA | | |
| | WC | |
| | C | |
| WA | | |
| C | | |
| | | RA |
| | | WB |
| | | WA |
| | | C |

- **Strictness:** This schedule is strict.
- **Rigorousness:** This schedule is not strict, as the transactions read values written by one another before the initially-reading transaction has committed.
- **Recoverability:** This schedule is recoverable as no transaction commits having read a value written by another, uncommitted transaction.
- **Strong Recoverability:** This schedule is strongly recoverable as it is strict.
- **Cascadability:** This schedule is cascadeless as no transaction reads a value written by an uncommitted transaction.

**b) Schedule T:**

| T1 | T2 | T3 |
|---|---|---|
| | RC | |
| | | RB |
| | RB | |
| RC | | |
| RA | | |
| | WC | |
| | C | |
| WA | | |
| | | RA |
| | | WB |
| C | | |
| | | WA |
| | | C |

- **Strictness:** This schedule is not strict because T3 reads value A, written by T1 before T1 commits.
- **Rigorousness:** This schedule is not rigorous because it is not strict. In a rigorous schedule, no transaction reads a value till other transactions which have read or written that value have committed.
- **Recoverability:** This schedule is recoverable as all of the transactions which read a value written by another transaction commit after the writing transaction has committed.
- **Strong Recoverability:** The schedule is not strongly recoverable as it is not strict.
- **Cascadability:** This schedule has the potential to cascade as T3 could read value A written by T1 before T1 commits. If T1 were to roll back after T3 read value A, T3 would also have to roll back.

**8. Serializability**

| **T1** | **T2** | **T3** |
|---|---|---|
| RB | | |
| | RA | |
| | | RB |
| RA | | |
| | RC | |
| | | RC |
| WB | | |
| | WA | |
| | | WC |
| | WC | |

a) **View Serializable?:** Yes.

b) **Conflict Serializable?:** No.

c) **Serializable Schedules:** This schedule is view serializable to T1, T3, T2; T3, T2, T1; or T3, T1, T2

**9. Sorting**

a) **Passes Needed:** 16 sort runs, 1 merge pass

**10. Bad Reads**

a) **Dirty Read:**

- **Description:** A dirty read is when a transaction reads a value which was written by another transaction. If the other transaction is rolled back, the first transaction will have read an inconsistent value.
- **Example:** Value of $x = 1$. T2 writes $x$ to be 2. T1 reads $x$ as 2. T2 rolls back. Value of $x$ is returned to 1.
- **Avoidance:** Ensure that all transaction which read a value written by another transaction wait to commit until after the writing transaction has committed.

b) **Unrepeatable Read:**

- **Description:** An unrepeatable read occurs when a transaction reads a value, a second

transaction over-writes that value, and then the first transaction reads the value again, getting two different answers.

- **Example:** T1 reads *x*, which is equal to 1. T2 writes *x* as 2. T1 reads *x* again, retrieving a different value from the first read.
- **Avoidance:** Unrepeatable reads can be avoided by preventing any transaction from reading or writing a value while another transaction has a lock on that value.

**11. Transaction Logs:** Transaction logs are useful because they facilitate recoverability, allowing a database to return to a consistent state in case of interrupted operation.

**12. MVCC**

**a) Description:** MVCC stands for multiversion concurrency control and consists of procedures carried about by database management systems in order to support concurrent access to databases.

**b) Example:** MVCC works by storing timestamped snapshots of a database state in order to provide each transaction operating on the system with a consistent, isolated version of the database. For example, if transaction one (T1) reads value *x* from the database and transaction two (T2) subsequently writes to value *x*, T1 will continue to view the original value of *x*.

**13. Queries**

**a) Find all planes flown by Pilot Bob stored at the hangar located at RDU.**

- **Relational Algebra:**

$$\pi_{Reg\#}\Big(\big(\big((\sigma_{Name = 'Bob'}(PERSON)) \bowtie_{Ssn=Ssn} PILOT\big) \bowtie_{Pid = Pid} FLIES\big) \bowtie_{Model = Model} \big((\sigma_{Location = 'RDU'}(HANGAR)) \bowtie_{Number = Hangar\_num} AIRPLANE\big) \bowtie_{Model = Model} PLANE\_TYPE\big)\Big)$$

- **SQL:**

```
SELECT AIRPLANE.Reg#
FROM AIRPLANE,
HANGAR,
PERSON,
PILOT,
FLIES,
PLANE_TYPE
WHERE PERSON.Name = 'Bob',
AND HANGAR.Location = 'RDU'
AND PERSON.Ssn = PILOT.Ssn
AND PILOT.Pid = FLIES.Pid
AND FLIES.Model = PLANE_TYPE.Model
AND PLANE_TYPE.Model = AIRPLANE.Model
AND AIRPLANE.Hangar_num = HANGAR.Number;
```

**b) Find all pilots who also own their planes.**

- **Relational Algebra:**

$$\pi_{Lic\_num}($$

$\pi_{Reg\#,\ Pid}$
(
AIRPLANE $\bowtie_{Model = Model}$ PLANE_TYPE $\bowtie_{Model = Model}$ FLIES
)
$\bowtie_{Reg\# = Reg\#}$
(
$\pi_{Reg\#,\ Ssn}$
(
AIRPLANE $\bowtie_{Reg\# = Reg\#}$ OWNS $\bowtie_{Id = Id}$ OWNER $\bowtie_{Ssn = Ssn}$ PERSON
)
)
)

- **SQL:**

```
SELECT PILOT.Lic_num
FROM PILOT,
(
        SELECT AIRPLANE.Reg#, FLIES.Pid
        FROM AIRPLANE, PLANE_TYPE, FLIES
        WHERE AIRPLANE.Model = PLANE_TYPE.Model
        AND PLANE_TYPE. Model = FLIES.Model
) AS TEMP1,
(
        SELECT AIRPLANE.Reg#, PERSON.Ssn
        FROM PERSON, OWNER, OWNS, AIRPLANE
        WHERE AIRPLANE.Reg# = OWNS.Reg#
        AND OWNS.Owner_id = OWNER.Owner_id
        AND OWNER.Ssn = PERSON.Ssn
) AS TEMP2,
WHERE TEMP1.Reg# = TEMP2.Reg#
AND PILOT.Pid = TEMP1.Ssn
AND PILOT.Pid = TEMP2.Pid;
```

**c) Find all pilots who have flown every type of plane.**

- **Relational Algebra:**
- **SQL:**

```
SELECT PILOT.Lic_num
FROM PILOT,
(
        SELECT FLIES.Pid
        FROM FLIES
        GROUP BY Pid HAVING COUNT(*) =
        (
                SELECT COUNT(DISTINCT PLANE_TYPE.Model)
                FROM PLANE_TYPE
        )
) AS TEMP
WHERE PILOT.Pid = TEMP.Pid;
```

**14. Functional Dependencies**

**a) Candidate Keys:**

- ABCDEFGH
- A
- G

- CE
- CD
- BC

**b) Decomposition into 3NF:**

- ABCDEFGH...

**15. Postgres**

**a)**

- ***tcp_keepalives* parameters:**
  - *tcp_keepalives_count*
  - *tcp_keepalives_idle*
  - *tcp_keepalives_interval*
- **Used for:** These values are used for automatically dropping external networked connections to the database if they time out (e.g. Have network problems). This functionality could be useful to prevent unnecessary deadlock situations.
- **Default Values:** By default, these three values are set to 0.
- **Used for Local Connections:** No.
- **Used for Remote Connections:** Yes.

**b) WAL**

- **Description:** WAL stands for "write ahead log" and represents ways to support the atomicity and durability of a database. WAL processes create a log of redo and undo information before any data modifications are carried out.
- **Uses:** WAL processes support database recoverability and replication.
- **Important Parameters:**
  - *checkpoint_segments* = 3
  - *checkpoint_timeout* = 5 minutes
  - *full_page_writes* = on
  - *checkpoint_warning* = 30s
- **Most Important:** It seems that the checkpoint warning setting may be the most important, even though it doesn't change the function of Postgres' WAL operations. It can be used to determine if your other settings are badly tuned, however. If checkpoint warnings are output often, it can indicate that the *checkpoint_segments* setting may be too low, resulting in excessive WAL writing.