

CS 178: Database Systems

B. NARAHARI

Department of Computer Science
GWU

Chapter 1

Query Processing and Optimization

CS 178: Database Systems

1.1 Query Processing: Introduction

- Consider the following relations:

PASSENGER (NAME, SSN, FLT_ID, MILES)

FLIGHT (FLT_ID, FLT_NO, STARTAPT, ENDAPT)

AIRPORT (APT, NAME, CITY)

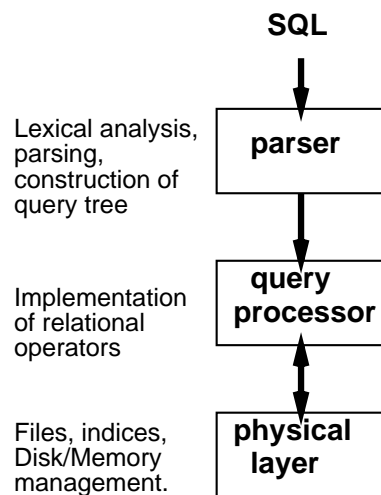
and the following query: “List passengers flying into National airport that have at least 1000 miles along with their mileage”.

In SQL:

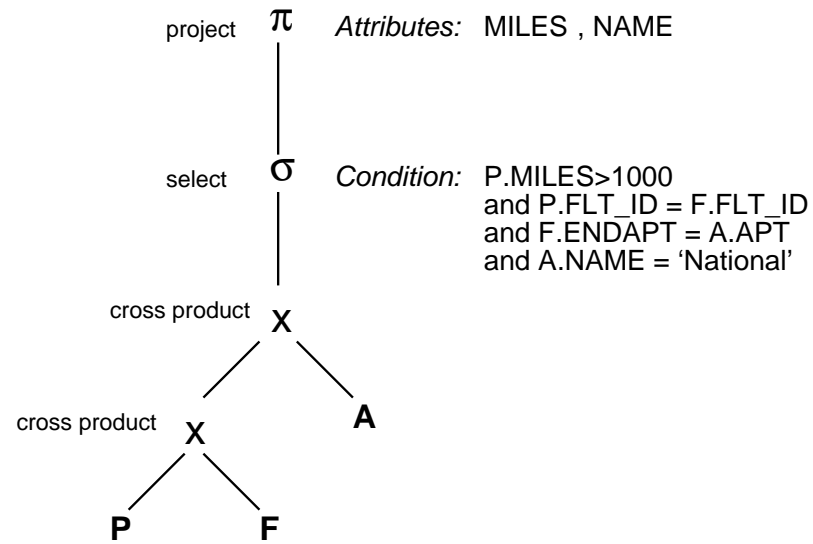
```
select P.NAME, P.MILES
from PASSENGER P, FLIGHT F, AIRPORT A
where P.MILES > 1000
and P.FLT_ID = F.FLT_ID
```

and F.ENDAPT = A.APT
and A.NAME = 'National';

An SQL query is parsed by the *parser* to create a *query tree*, which is then executed by the *query processor*.



A typical query tree produced by the parser:



NOTE:

- There are several ways of executing the same query tree.
 - An *execution plan* specifies an order.
 - A parser provides a simple tree.
 - The query processor creates a better tree and an execution plan.
- For our example, assume the following sizes (with 1 Kb block-size)

PASSENGER 50 bytes per tuple (20 tuples per block)

100,000 tuples (5000 blocks)

FLIGHT 20 bytes per tuple (50 tuples per block)

50,000 tuples (1000 blocks)

Assume: each block access takes 20 ms (milliseconds).

For example, a scan of PASSENGER requires $5000 \times 20 \text{ ms} = 100 \text{ seconds}$.

Implementing Selection

- First consider a single equality search, e.g.,

$\sigma_{\text{NAME}='Smith'}$ (PASSENGER)

1. Linear search on a heapfile:

- 5000 blocks scanned (worst-case) \Rightarrow 100 seconds.
- If NAME is a *key*: \Rightarrow 2500 blocks (average) \Rightarrow 50 seconds.

2. Binary search on a sorted file (sorted by NAME):

- Binary search takes $\lceil \log_2 n \rceil$ for a file of n blocks. $\Rightarrow \lceil \log_2 5000 \rceil$ blocks \Rightarrow 13 blocks $\Rightarrow 13 \times 20$ ms \Rightarrow 260 ms.

3. B+-tree index on NAME:

- Suppose 100 index entries fit into a block $\Rightarrow 2m - 1 = 100$ ($m = \text{degree}$) $\Rightarrow m = 50$.
- Leaf level has as many entries as tuples \Rightarrow 100,000 entries \Rightarrow 1000 leaf blocks (best-case: all packed) \Rightarrow 2000 leaf

blocks (worst-case)

- How many B+-tree levels needed (worst-case)? \Rightarrow Recall:
 $2m^{k-1}$ at level $k \Rightarrow$ we want least value of k such that
 $2m^{k-1} \geq 2000 \Rightarrow k = 3$ (levels 0,1,2,3) \Rightarrow 4 levels
- Access time? \Rightarrow 5 accesses in all (data block requires 1 access) $\Rightarrow 5 \times 20$ ms \Rightarrow 100 ms.

4. Hash index on NAME:

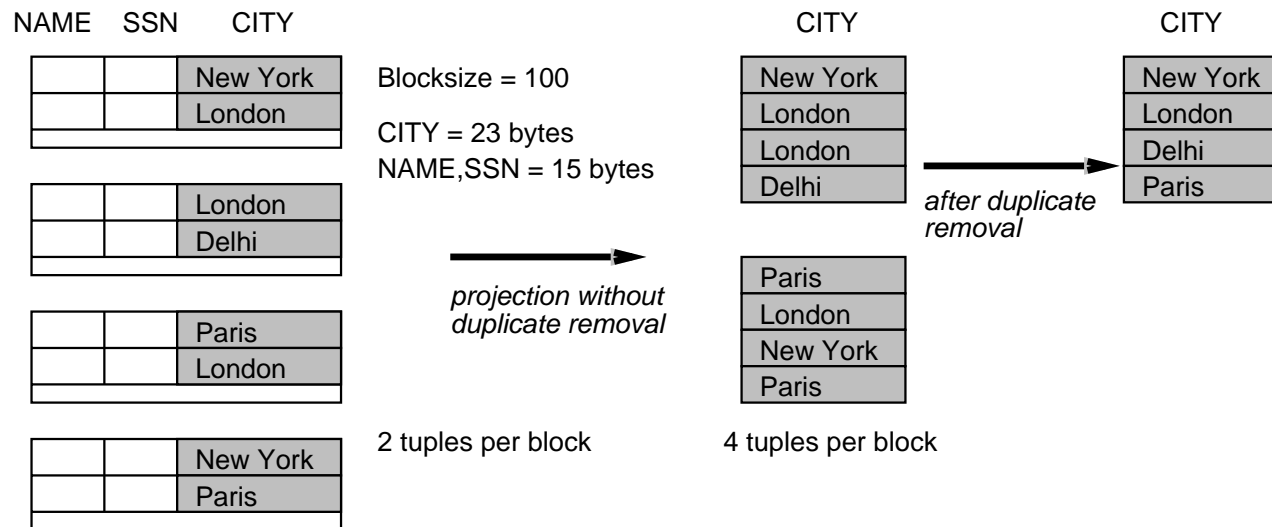
- Assume each chain has 2 blocks (1 overflow block). \Rightarrow 1 block access for directory, 2 for data \Rightarrow 3 block accesses $\Rightarrow 3 \times 20$ ms \Rightarrow 60 ms.

- Selects with multiple predicates - CNF representation
conjunctive normal form: set of conjunctions/ANDs, each is a set of ORs
- Selection plan depends on indices available and file organization

Implementing Projection

- Two actions are required in implementing projection:
 1. Scan through relation and extract desired attributes.
 2. Remove duplicates.

Example: Consider EMP (NAME, SSN, CITY) and the query $\Pi_{\text{CITY}} (\text{EMP})$:



- Observe:
 - Step 1 (projection) is straightforward.
 - Step 2 (duplicate removal) requires more work.
 - Often, parts of Step 2 are integrated into Step 1.
 - Sometimes duplicate removal is not needed (if not specified with a **distinct** in SQL).
- Duplicate removal via sorting

- Duplicate removal using Hashing:

- Study Notes for more details on algorithms for Selection and Projection
- Focus now on Join algorithms
 - time for Joins is an order of magnitude larger

Implementing Joins

- Joins take up significant time
- occur frequently \Rightarrow result of normalization
- Joins vs. cross-products:
 - Every join can be expressed as a cross-product, e.g., $PASSENGER * FLIGHT$ is the same as $\sigma_{PASSENGER.FLT_ID=FLIGHT.FLIGHT}$
 - SQL parsers typically only report cross-products.
 - Cross-products are large \Rightarrow system should recognize a join where possible.

- We will use the following example: PASSENGER * FLIGHT where

PASSENGER has

$n_P = 5000$ blocks,

20 tuples per block

$r_P = 100,000$ tuples

FLIGHT has

$n_F = 1000$ blocks

50 tuples per block

$r_F = 50,000$ tuples

Join Algorithms

- Several ways to implement a join:
 1. Simple nested loops.
 2. Block-nested loops.
 3. Nested loops with indices.
 4. Sort-merge join.
 5. Hash-join.

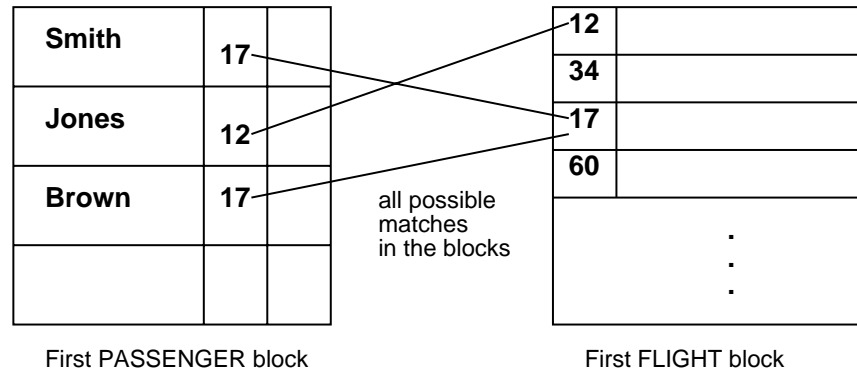
- Simple nested loops.
 - Scan tuples in one file and for each one, scan the other file to find matching tuples.
 - For example, suppose we scan PASSENGER in the outer loop:

Simple Nested Loops

1. **for each** tuple $x \in$ PASSENGER
2. **for each** tuple $y \in$ FLIGHT
3. **if** $x.FLT_ID = y.FLT_ID$
4. put joined tuple in result

- What is the cost?
 $\Rightarrow (r_P \times n_F) + n_P = 10^8 + 5000$ blocks overall
 $\Rightarrow 555.58$ hours!
- What if FLIGHT were the outer relation?
 $\Rightarrow (r_F \times n_P) + n_F$ overall
 $\Rightarrow 25 \times 10^8 + 1000$ blocks overall $\Rightarrow 1388.89$ hours!
 \Rightarrow it's worse!
- Using simple nested loops is a really stupid way of implementing a join.

- Block-nested loops:
 - iterate “block by block” instead of iterating over tuples
 - Consider reading the first block of PASSENGER.
 - When we read the first block of FLIGHT, we can join all possible combinations in each block:



- Then, we read the next block of FLIGHT and find all possible matches in the first block of PASSENGER. Then, the next block of FLIGHT ... and so on until all blocks of FLIGHT are joined with the
- Then, the next block of PASSENGER is read and joined with the first, then second, then third ... etc blocks of FLIGHT.
- ... And so on until all of PASSENGER has been scanned.

Block-Nested Loops

1. **for each** block \in PASSENGER
2. **for each** block \in FLIGHT
3. if a pair in each block matches
4. put joined tuple in result

- Why does this work? \Rightarrow each pair of tuples needs to be tried against each other only once, block at a time.
- Cost: $\Rightarrow (n_P * n_F)$ iterations
 $\Rightarrow (n_P * n_F) + n_P$ overall time
 $\Rightarrow 5000 \times 1000 + 5000$ blocks $\Rightarrow 5,005,000$ blocks overall
 $\Rightarrow 27.78$ hours
- How much memory has been used? $\Rightarrow 2$ input buffers (1 for each file).
- Can we do better with more buffers?

- Example: suppose we have $M = 100$ input buffers. \Rightarrow use K buffers for PASSENGER and $M - K$ for FLIGHT. Suppose PASSENGER is outer relation:
 - Read first K blocks of PASSENGER.
 - Scan through all of FLIGHT and match tuples.
 - Read next K blocks of PASSENGER.
 - Scan through all of FLIGHT and match tuples.
 - ... and so on.
- Cost? $\Rightarrow \frac{n_P}{K}$ groups of PASSENGER (exact: $\lceil \frac{n_P}{K} \rceil$) $\Rightarrow \lceil \frac{n_P}{K} \rceil$ scans of FLIGHT $\Rightarrow \lceil \frac{n_P}{K} \rceil * n_F$ blocks \Rightarrow totally $n_P + \lceil \frac{n_P}{K} \rceil * n_F$
- What choice of K is optimal? \Rightarrow as large as possible: $K = M - 1$
- In our example: $\Rightarrow 5000 + \lceil \frac{5000}{99} \rceil * 1000$ blocks $\Rightarrow 56000$ blocks

overall $\Rightarrow 56000 \times 20 \text{ ms} \Rightarrow 18.6 \text{ minutes} \Rightarrow$ a great improvement!

- Can we do better? \Rightarrow use FLIGHT as outer relation $\Rightarrow 1000 + \lceil \frac{1000}{99} \rceil * 5000$ blocks $\Rightarrow 56000$ blocks overall \Rightarrow same as before.
- Note: suppose FLIGHT had 90 blocks: With PASSENGER outside: $\Rightarrow 5000 + \lceil \frac{5000}{99} \rceil * 90$ blocks $\Rightarrow 9590$ blocks. With FLIGHT outside: $\Rightarrow 90 + \lceil \frac{90}{99} \rceil * 5000$ blocks $\Rightarrow 5090$ blocks (almost half the time) \Rightarrow use smaller relation in outer loop.

Nested loops with indices

- If there's an index on the join attribute, we can use it.
- Example: suppose we have an index on PASSENGER.
- Use PASSENGER as inner relation:

Nested Loops with Indices

1. **for each** block in FLIGHT
2. **for each** tuple in the current block
3. extract key;
4. search for corresponding tuple in PASSENGER using
5. place joined tuple if match occurs
6. **endfor**

- Cost:

- FLIGHT is scanned once $\Rightarrow n_F = 1000$ blocks.
 - For each *tuple* of FLIGHT, we probe index $\Rightarrow r_F \times$ number of blocks per access
 - The actual cost depends on the type of index.
1. B+-tree index for PASSENGER: $\Rightarrow 5$ blocks per access
 $\Rightarrow 50000 \times 5$ blocks of PASSENGER $\Rightarrow 251,000$ blocks overall $\Rightarrow 1.39$ hours.
 2. Hash index for PASSENGER: \Rightarrow about 2 blocks per access
 $\Rightarrow 101,000$ blocks overall $\Rightarrow 33.7$ minutes.

In general if α_P blocks are required by the index, then the total cost is

$n_F + r_F * \alpha_P$ blocks.

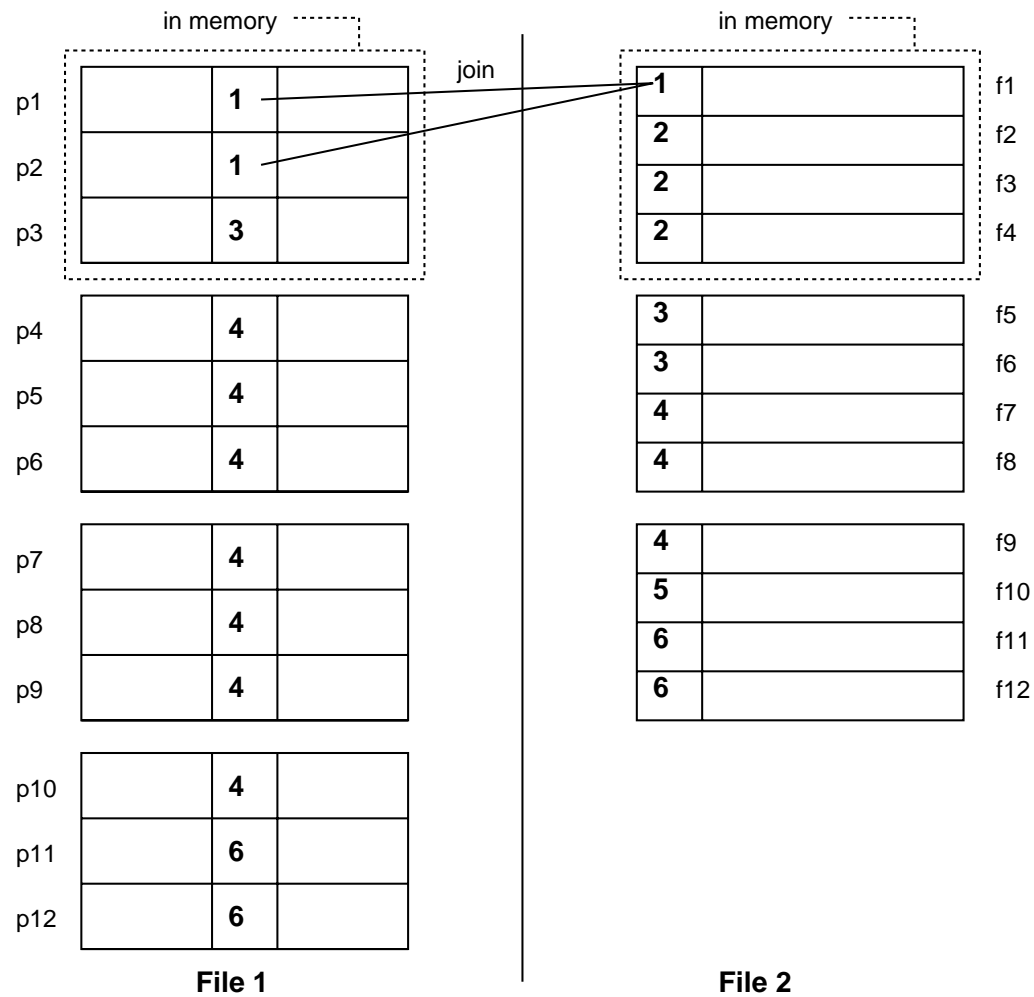
Sort-merge Join

- Key idea:
 - Sort PASSENGER.
 - Sort FLIGHT.
 - Set up sorted files as if they were going to be merged.
 - Process a merge, but only produce as output joined tuples.
- How to merge?
 - Read first block of PASSENGER.
 - Read first block of FLIGHT.
 - Match all the tuples that can be matched.
 - Suppose last tuple of PASSENGER block has FLT_ID=6.
 - Suppose last tuple of FLIGHT block has FLT_ID=13.

– Are there any other tuples in FLIGHT that need to be matched with FLT_ID=6? \Rightarrow No! \Rightarrow Don't need to look further in FLIGHT.

- Example of sort-merge join:

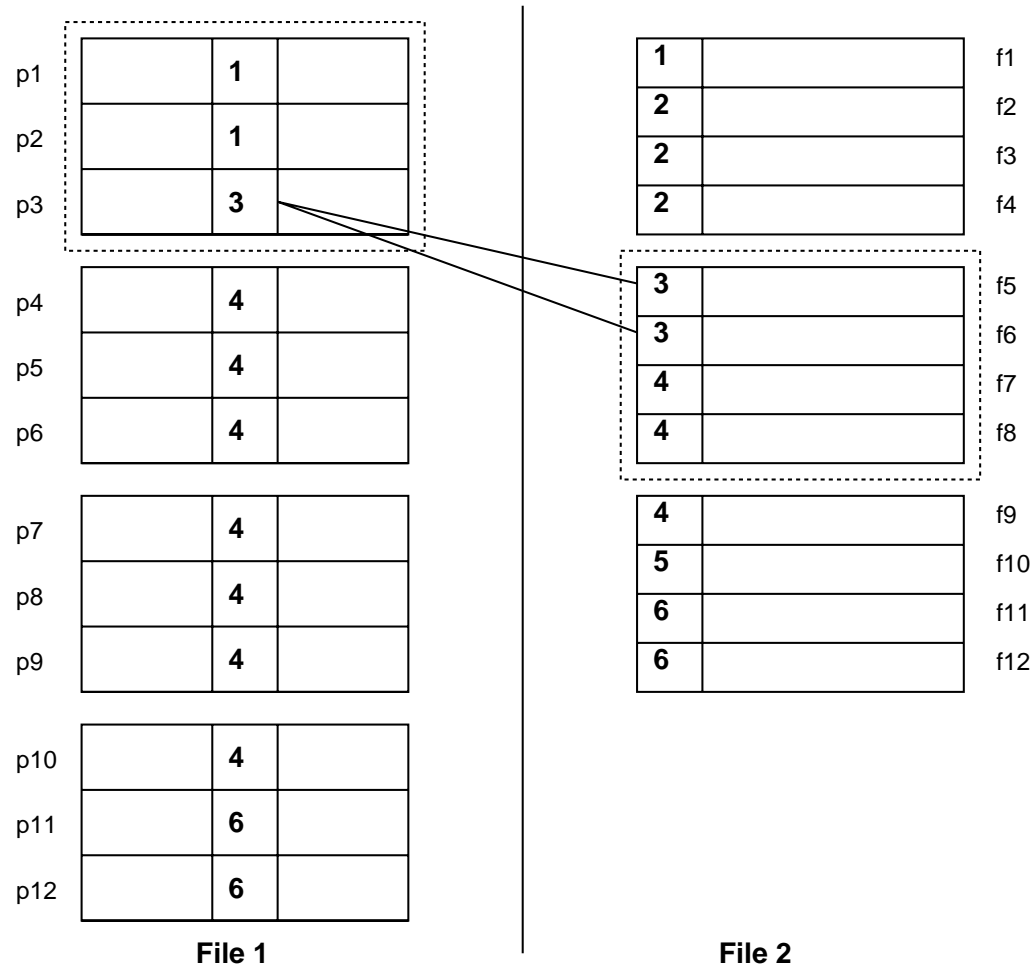
Recall the relations: PASSENGER (NAME, SSN, FLT_ID, MILES) and FLIGHT (FLT_ID, FLT_NO, START_APT, END_APT).



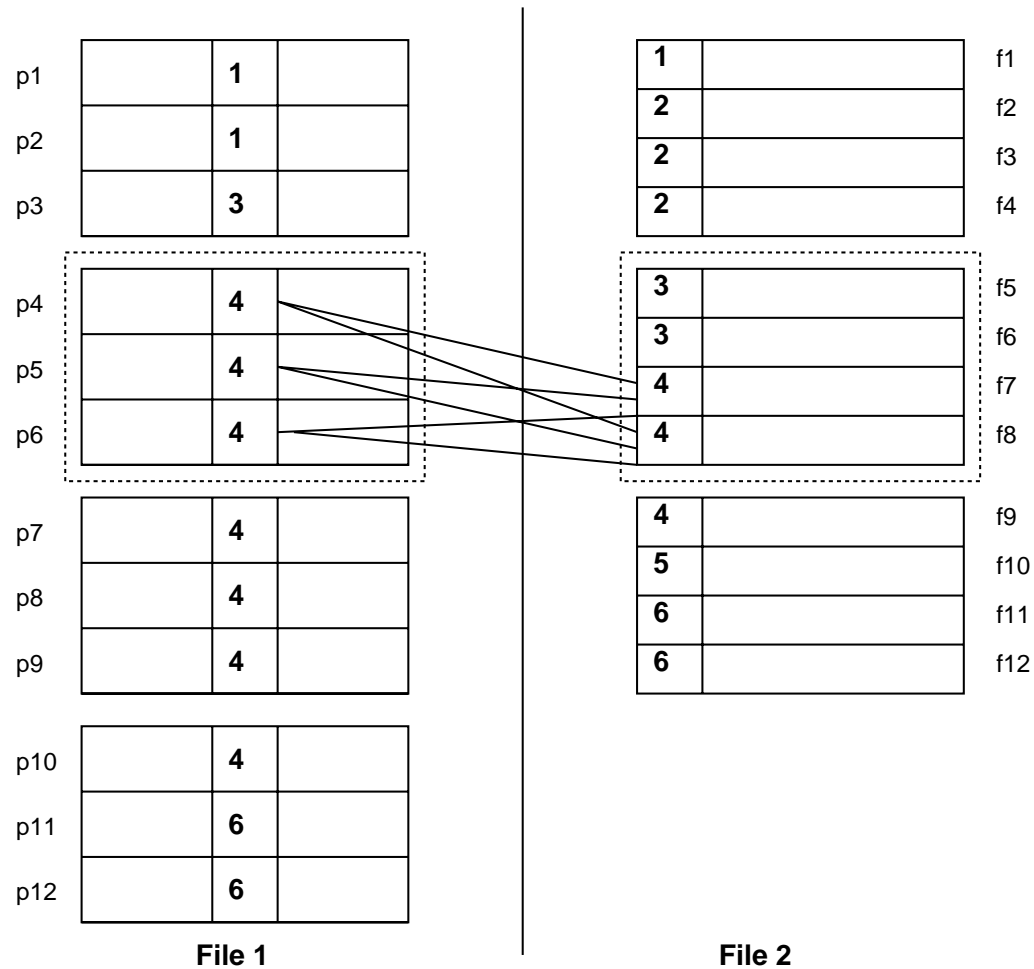
Let p_1, p_2, \dots denote the PASSENGER tuples and f_1, f_2, \dots denote the FLIGHT tuples.

1. First, we output $\langle p1, f1 \rangle, \langle p2, f1 \rangle$.

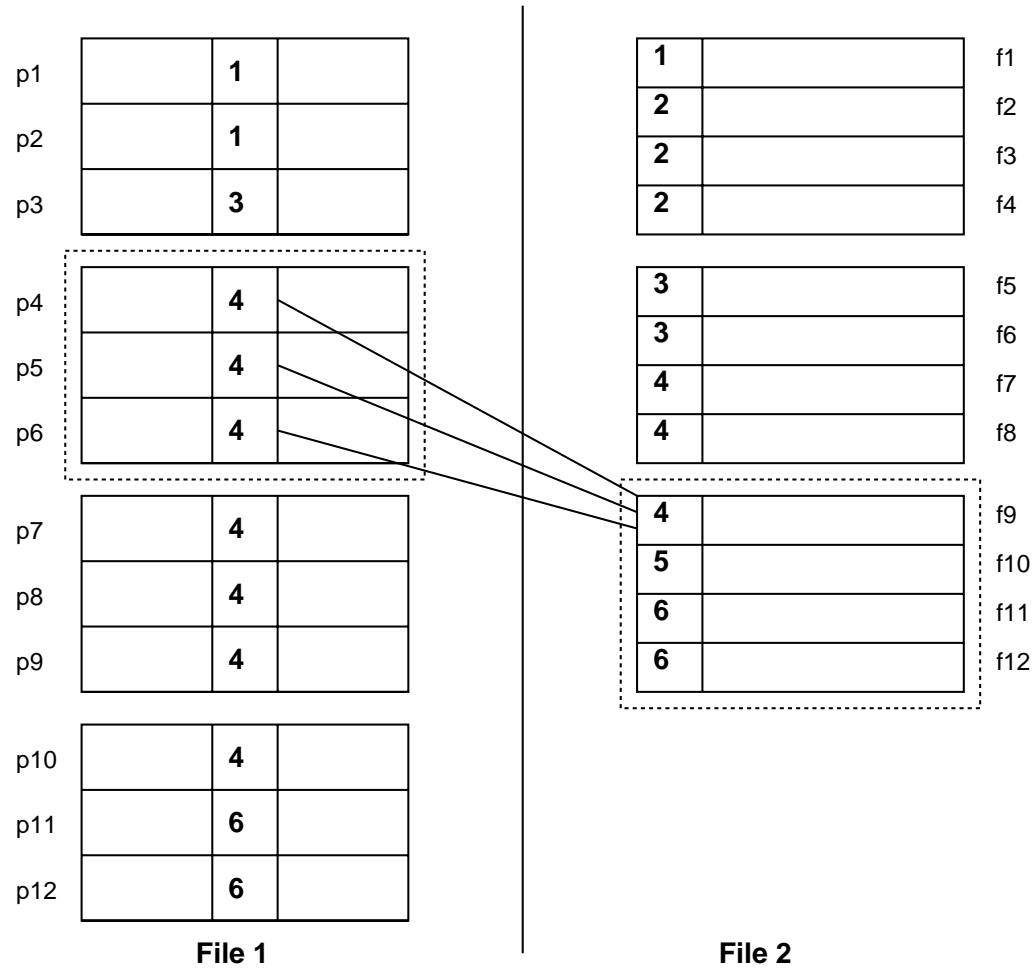
2. Then, read the second FLIGHT block and output $\langle p3, f5 \rangle$, $\langle p3, f6 \rangle$:



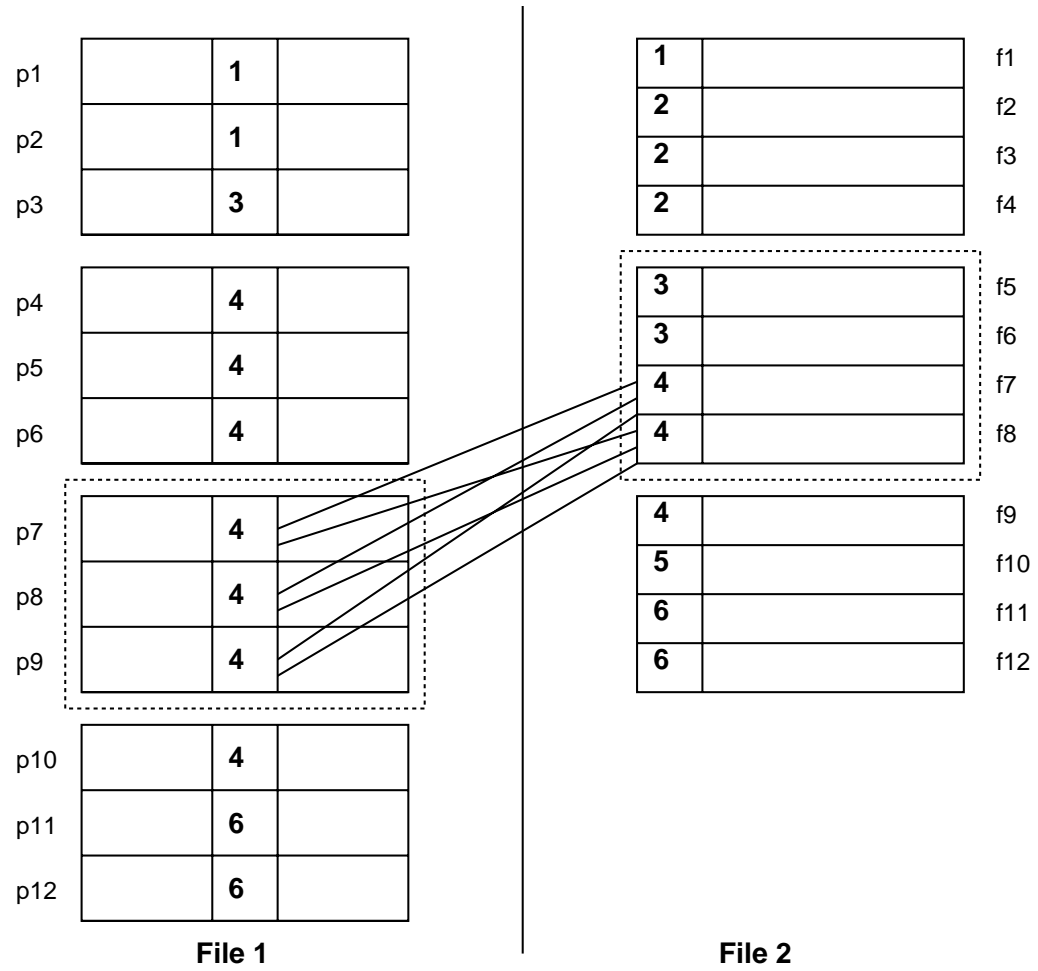
3. Read the second PASSENGER block and output $\langle p4, f7 \rangle$, $\langle p4, f8 \rangle$, $\langle p5, f7 \rangle$, $\langle p5, f8 \rangle$, $\langle p6, f7 \rangle$, $\langle p6, f8 \rangle$.



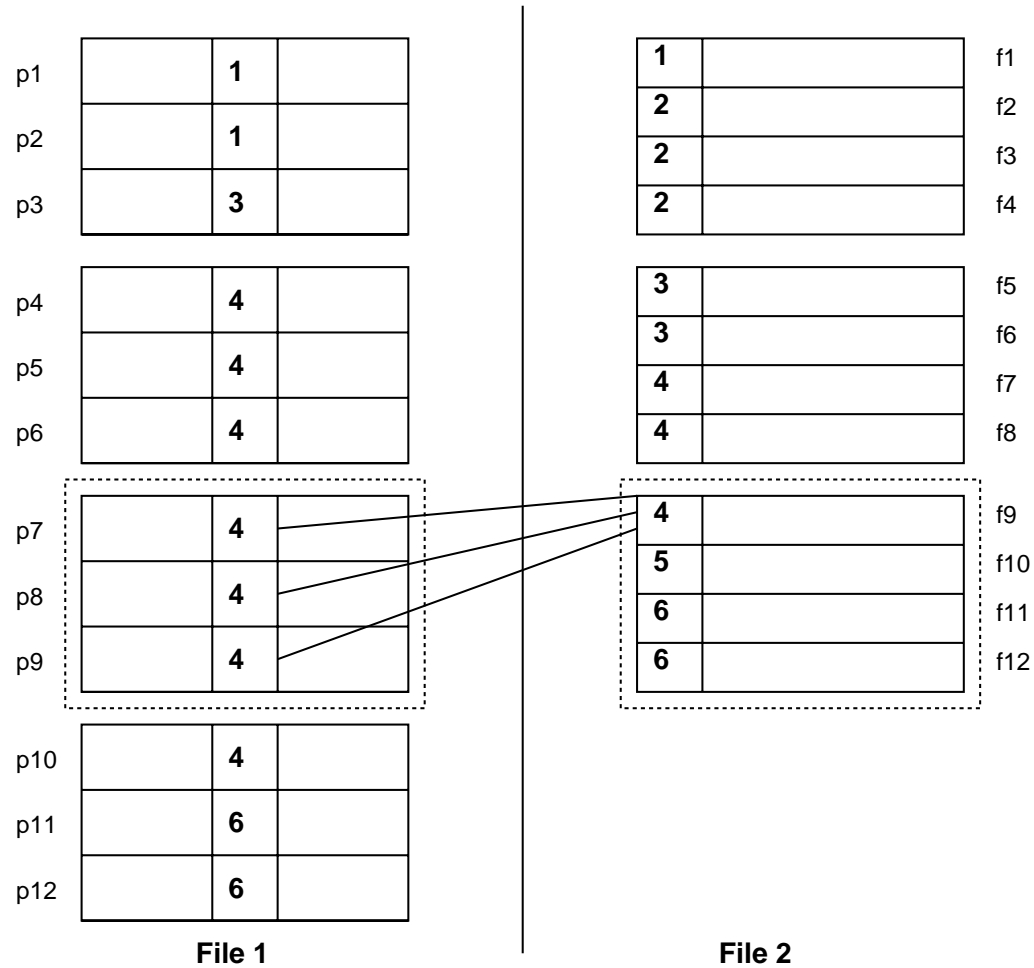
4. Read next FLIGHT block and output $\langle p4, f9 \rangle$, $\langle p5, f9 \rangle$, $\langle p6, f9 \rangle$.



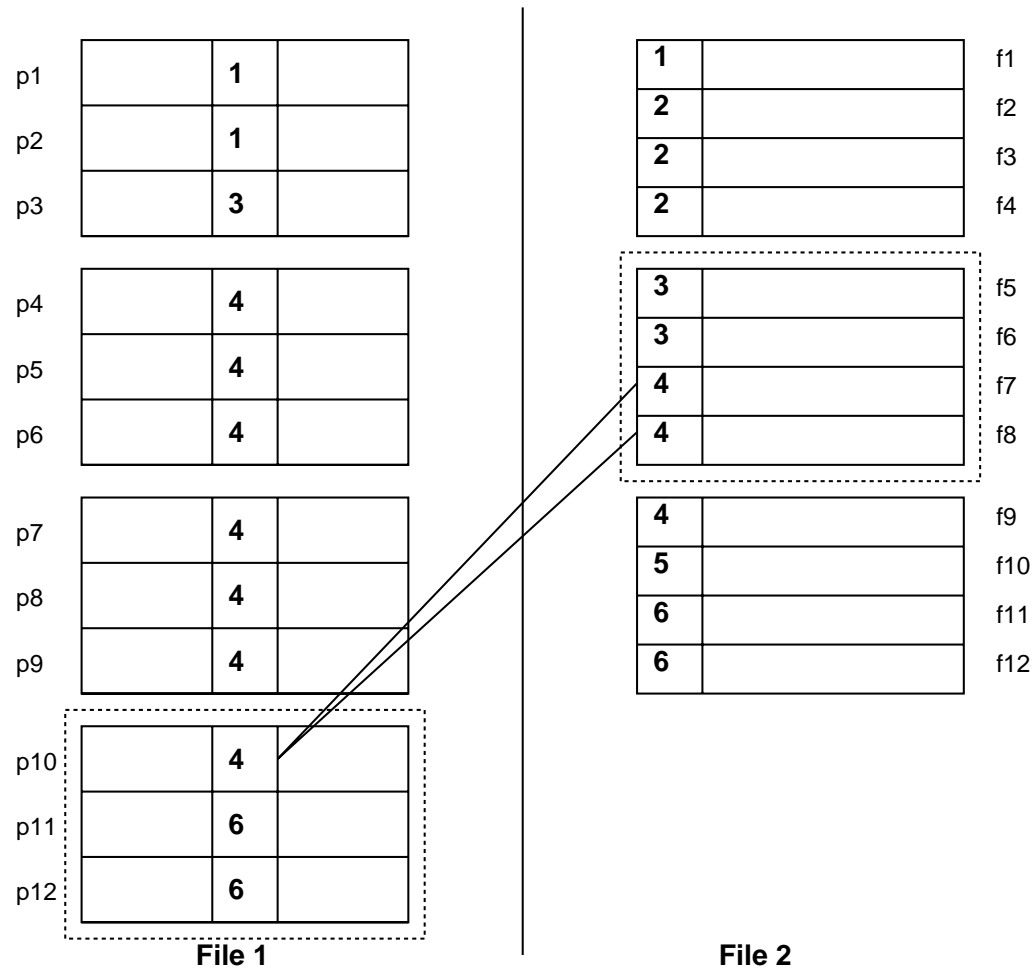
5. Read next PASSENGER block \Rightarrow the item '4' continues
 \Rightarrow must go back to start of '4' in FLIGHT \Rightarrow a reversal!
Output $\langle p7, f7 \rangle$, $\langle p7, f8 \rangle$, $\langle p8, f7 \rangle$, $\langle p8, f8 \rangle$, $\langle p9, f7 \rangle$,
 $\langle p9, f8 \rangle$.



6. Read next FLIGHT block and output $\langle p7, f9 \rangle$, $\langle p8, f9 \rangle$, $\langle p9, f9 \rangle$.



7. Read next PASSENGER block \Rightarrow '4' continues \Rightarrow must go back to first '4' in FLIGHT \Rightarrow another reversal!
 Output $\langle p10, f7 \rangle, \langle p10, f8 \rangle$.



- Cost:

- If there are enough buffers \Rightarrow reversal not needed.
- Typically, reversal does not occur often \Rightarrow only one scan through each sorted file.
- Time needed for sorting: $2n_P \log_M n_P + 2n_F \log_M n_F$.
- Time needed for merging: $n_P + n_F$. \Rightarrow Total: $2n_P \log_M n_P + 2n_F \log_M n_F + n_P + n_F$.
- Suppose $M = 100$ in our example: $\Rightarrow 2 \times 5000 \log_{100} 5000 + 2 \times 1000 \log_{100} 1000 + 5000 + 1000$ blocks $\Rightarrow 30,000$ blocks $\Rightarrow 10$ minutes (best so far).

- Note:

- Some runs of a particular FLT_ID could span several blocks \Rightarrow hopefully all fit into memory simultaneously.
- If ‘runs’ of a particular value occur \Rightarrow try to reserve memory

for these runs.

- It is possible to speed up the process:
 1. Sort both files simultaneously.
 2. During each merge of the sort process, perform joins.
- The sort-merge join is very general \Rightarrow it can be applied to any join condition.
- A join on multiple attributes can be handled by sorting on the combination.

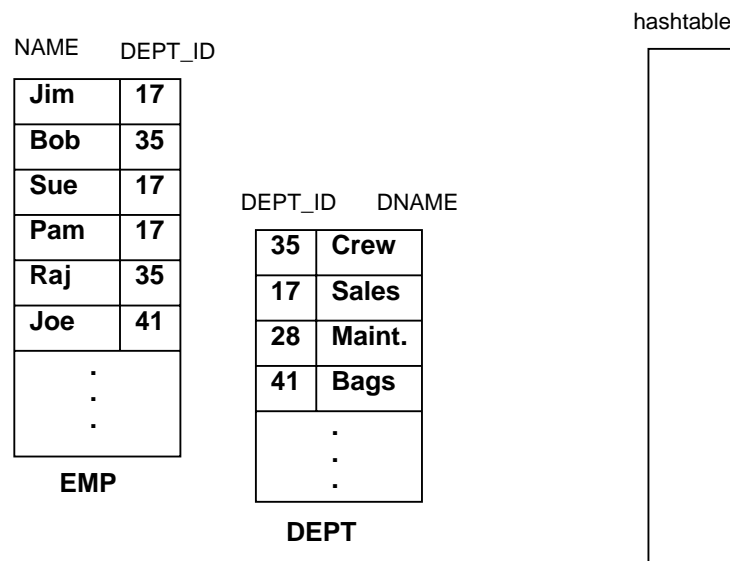
Hash joins:

- Key ideas:
 - two phase algorithm:
 - * phase 1: hash the two relations
 - * phase 2: create the joined tuples
 - Use a single hash function on the join attributes.
 - Hash the first file into a hash file using the function.
 - Hash the second file into the *same* file using the same hash function.
 - Records that are candidates for joining must lie in the same bucket.
 - Process the hash file bucket by bucket.
 - Read in a bucket and search for matches.

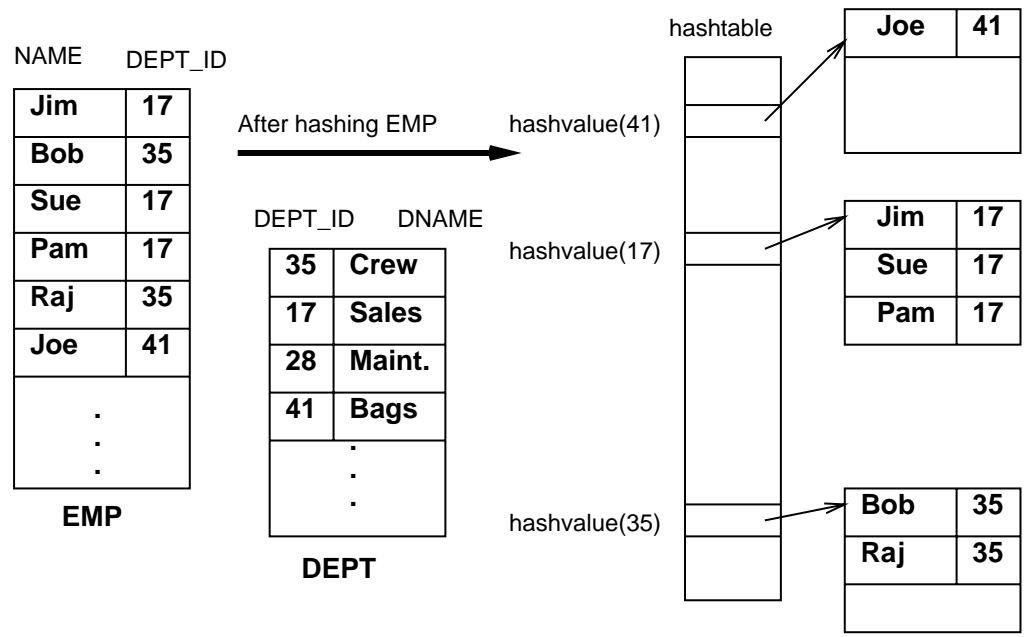
- Some details:
 - $2(n_F + n_P)$ time for Phase 1 IF we have enough pages to hold B buckets in memory
 - \Rightarrow read block, write block once
 - $(n_F + n_P)$ time for Phase 2
 - Total time for Hash Join $3(n_F + n_P)$ – *fastest* thus far...
 - what is the catch?
 - \Rightarrow algorithm's effectiveness depends on size of main memory

- If we use too many buckets, we will be doing a lot of I/O.
 \Rightarrow worst-case: 2 I/O's per tuple inserted \Rightarrow inserting 100,000 tuples means 100,000 block accesses!
 (at least one block access per *tuple* inserted). \Rightarrow Optimal number of buckets is M (memory size).
- Assuming uniform distribution across buckets \Rightarrow each bucket has $\frac{n_P+n_F}{M}$ blocks.
- To process a bucket, the bucket has to fit into memory.
 \Rightarrow better to have $\frac{n_P+n_F}{M} \leq M \Rightarrow M \geq \sqrt{n_P + n_F}$ ideally.
- It is not always possible to guarantee this condition \Rightarrow when condition does not hold then time increases

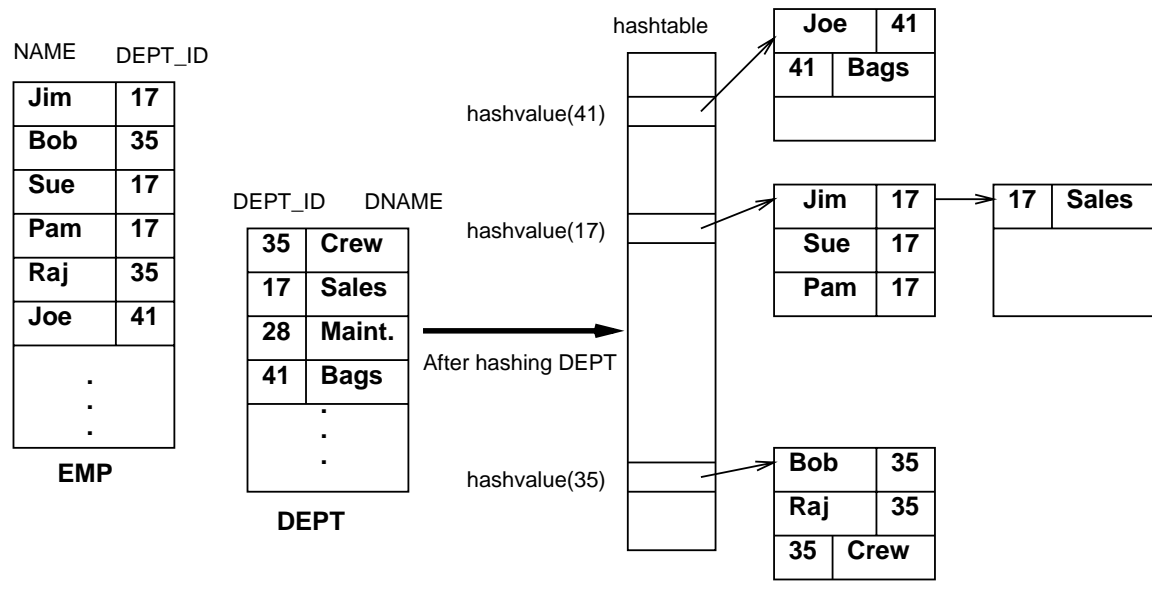
- An example: consider a join of EMP and DEPT on the attribute DEPT_ID.
Initially, the hashtable is empty:



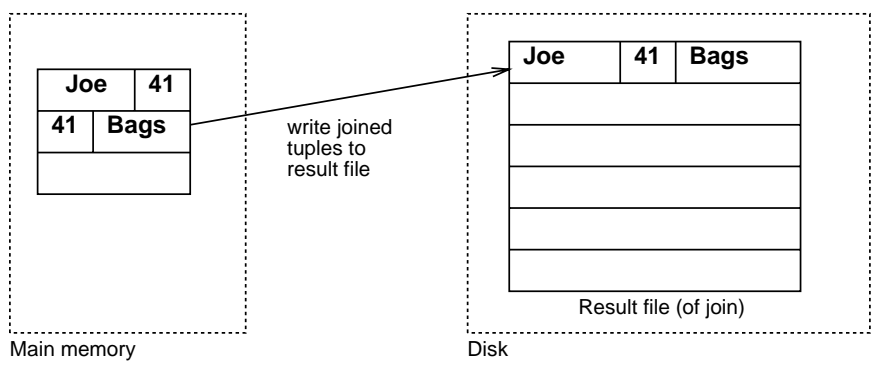
First, the file EMP is hashed (on DEPT_ID):



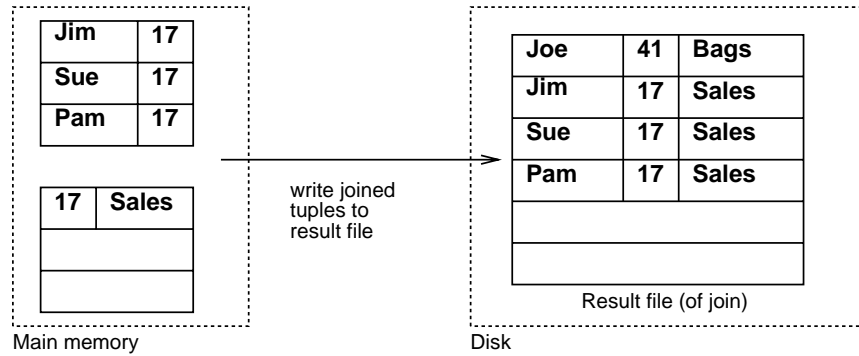
Then, the file DEPT is hashed into the same hash file:



Now, the buckets are processed one by one, e.g, bucket for '41':



Process bucket for '17':



And so on until all buckets have been processed.

- To process a large bucket:
 - Read in tuples from bucket and hash them to a temporary file.
 - Use a *different* hashing function than before.
 - Why? The same hashing function as before will only duplicate the bucket.
 - Again, process the new ‘sub-buckets’ one by one.
 - Hopefully the second hashing operation will create smaller buckets that fit into memory.
 - Worst-case, the procedure may have to be repeated.

Memory Management in a DBMS:

- The above discussion shows us why it’s better for a DBMS to manage memory by itself.

- In a block-nested join, once a block for the outer relation has been processed, it is not needed again \Rightarrow better to throw it out immediately and use the memory for the inner relation.
- In other methods, we have explicitly assumed fixed buffer sizes (M) \Rightarrow DBMS memory management.

Query Optimization: Overview

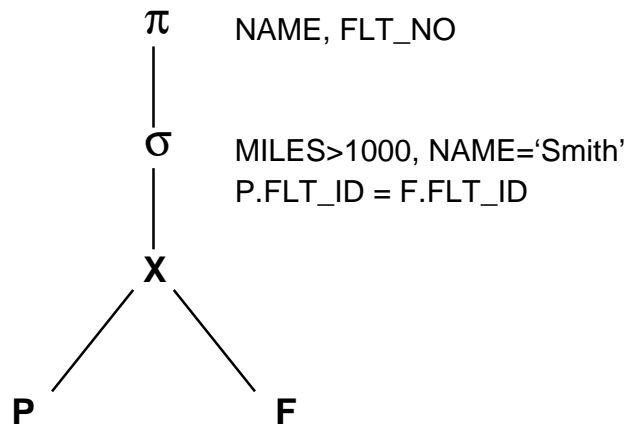
- Key functions in query optimization:
 - Enumerate several query plans.
 - Evaluate the cost of each plan.
 - Select the best such plan.
- Enumerating plans:
 - A query plan includes:
 - * A query tree.
 - * Methods assigned to implement operators (e.g., Sort-merge for a given join).
 - * An order of execution whenever different orders are possible.

- Plans are enumerated by considering alternative ways of handling a query.
 - * also depends on parameters such as size of memory, indices available etc.
 - * for example, hash join efficiency depends on a certain minimum memory size
- In general, enumeration is combinatorially explosive \Rightarrow heuristics must be considered.
- It helps to use heuristic rules for specific operators, e.g., *push select's past joins*.
- Evaluating the cost of a particular plan:
 - Use a metric such as the *number of block accesses* to evaluate I/O cost.

- Use techniques for cost estimation of relational operators (described earlier).
- Put together the total cost for a plan.

- Consider the following example: **ll select** NAME, FLT_NO
from PASSENGER P, FLIGHT F
where P.FLT_ID=F.FLT_ID
and MILES>1000
and NAME='Smith'

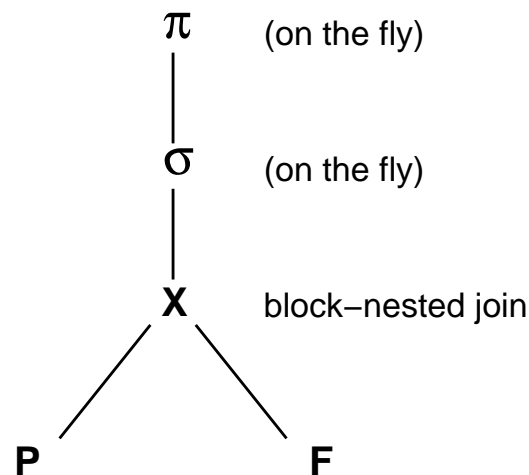
The query tree produced by the parser typically looks like:



- Several options are possible with the above query:

1. Operator methods:

- Use block-nested join with FLIGHT (F) as outer relation.
- Compute both σ and Π on the fly (as results are produced by the join)

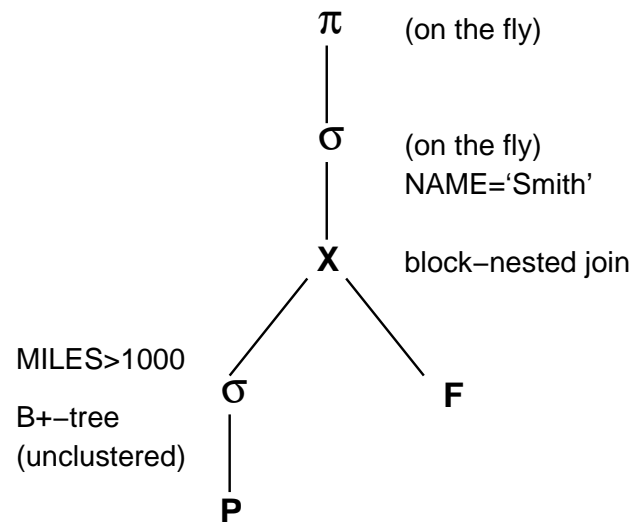


Cost estimation:

- Recall cost of block-nested join: $\Rightarrow n_F + \lceil \frac{n_F}{M-1} \rceil n_P$ blocks
 $\Rightarrow 1000 + \lceil \frac{5000}{99} \rceil 1000$ blocks $\Rightarrow 51506$ blocks.
- No cost for computing σ or Π .
- Total: 51506 blocks $\Rightarrow 51506 \times 20$ ms $\Rightarrow 17.1$ minutes.

2. Methods:

- Push the “MILES>1000” condition past the join to PASSENGER.
- Use B+-tree (unclustered, say) for select.
- Use block-nested join with FLIGHT as outer relation.
- Compute remaining operators on the fly.

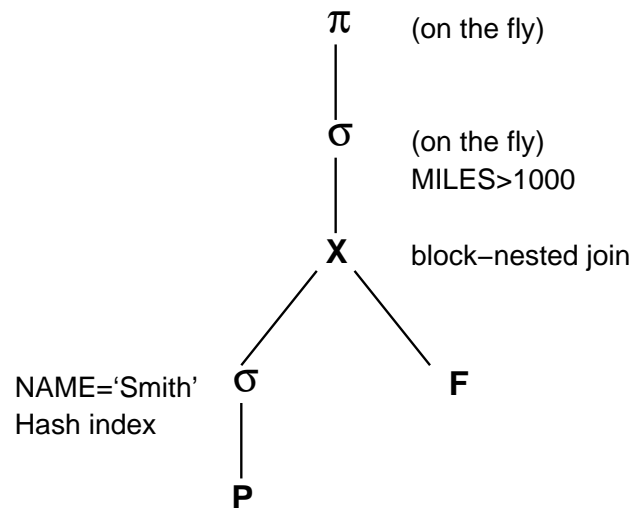


Cost:

- 2000 tuples (of PASSENGER) satisfy $MILES > 1000 \Rightarrow$ (worst-case) 2000 data blocks, 4+20 tree blocks (tree and leaf-level) \Rightarrow 2024 blocks accessed.
- 2000 tuples (of PASSENGER) satisfy $MILES > 1000 \Rightarrow$ 100 blocks produced as a result \Rightarrow these 100 blocks will be joined with FLIGHT's 1000 blocks.
- Block nested join with FLIGHT: $\Rightarrow 1000 + \lceil \frac{100}{99} \rceil 1000$ blocks \Rightarrow 3000 blocks.
- No cost for remaining select's and project.
- Total: 5024 blocks $\Rightarrow 5024 \times 20$ ms \Rightarrow 1.67 minutes.
- Note: we could use FLIGHT as inner relation $\Rightarrow 100 + \lceil \frac{100}{99} \rceil 1000$ blocks \Rightarrow 2100 blocks \Rightarrow 4124 blocks overall \Rightarrow 1.37 minutes.

3. Methods:

- Push the other select down ($\text{NAME}=\text{'Smith'}$).
- Assume hash index exists on NAME.
- Block nested join with FLIGHT as inner relation.



Cost:

- 45 tuples satisfy $\text{NAME}=\text{'Smith'}$ \Rightarrow 3 blocks \Rightarrow 4 blocks overall (including directory block).

- Block-nested join with FLIGHT as inner relation: $\Rightarrow 4 + \lceil \frac{4}{99} \rceil 1000$ blocks $\Rightarrow 1004$ blocks.
- Total: 1008 blocks $\Rightarrow 20.2$ seconds.

Manual Intervention – Tuning

- A DB Administrator (DBA) can take steps to improve the performance of some “tough” queries:
 - Decide to create or remove indices.
 - Decide types of indices.
 - Use tools or system options to re-organize data on disk.
 - A DBA needs to understand the system workload:
 - Frequently-used queries and how frequently they are used.
 - Updates and update frequencies.
 - User requirements (complaints).
- ⇒ identify the important queries (users?) and focus on them.

- It may be tempting to define an index on *every* attribute used in the important queries \Rightarrow indices make (read) access faster.

However, indices also:

- take up space (often as large as the relation itself) \Rightarrow drains virtual memory (code/data), clutters up disk (data)
- are bad for insertions \Rightarrow compare insertion using a B-tree versus insertion into a heapfile.

Fast updates are desirable in some applications, e.g., banking.

- General rules of thumb in index creation:
 - Create indices for attributes that occur in multiple queries.
 - Avoid creating indices for small relations.
 - If some updates are important, be careful about creating indices for attributes involved in the update.

Note: hash indices allow for fast insertion

- If a B+-tree index is defined on non-key attributes, consider using a clustered index.
- For equality selections, use a hash index provided range searches are not required.