MSiA-413 Introduction to Databases and Information Retrieval

Lecture 11
Indexing Databases

Instructor: Nikos Hardavellas

Slides adapted from Steve Tarzia

Last Lecture

Indexes stored in cache and it's very fast

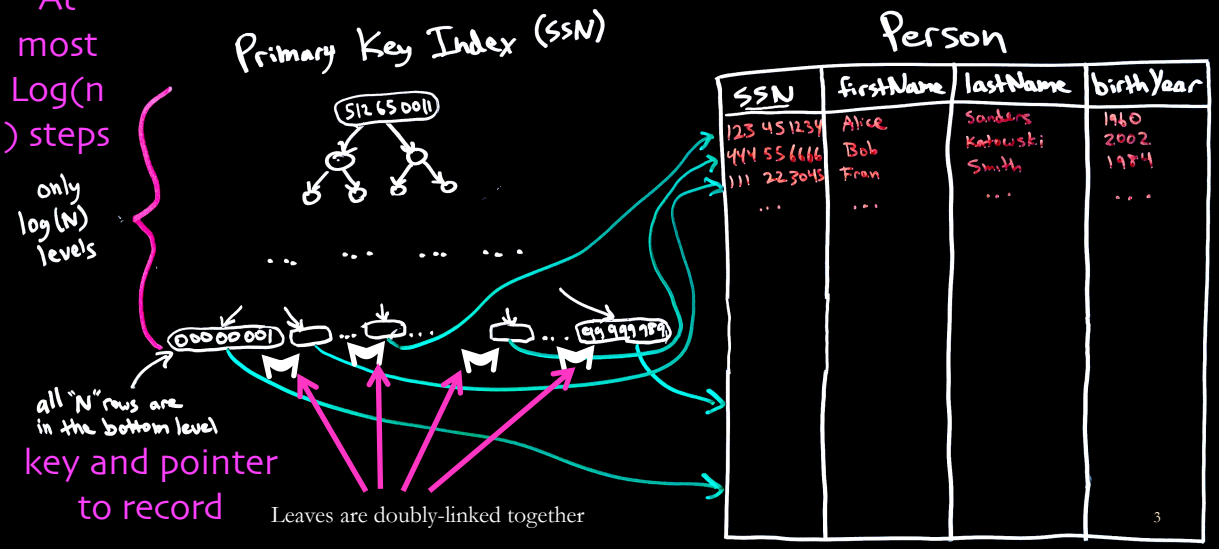
- Computer memory can be thought of as one big array
  - Expensive to move large chunks of data
  - Let data remain permanently in a memory *address*, refer to it with the address number
- A Tree (or graph in general) can be stored as a list of nodes referring to other nodes by memory address
- Trees can serve as an *index* to find data quickly in a database
  - Logarithmic #steps, tree can be memory-resident → fast access, no need to sort data
- Insertions and deletions in a tree are fast as well
  - Most of the data remains in place; we just change a few references
- Trees need to implement *balancing rules* during insertions and deletions
  - e.g., B^+-trees are self-balancing (will not cover self-balancing trees in this class)

Review of how an index finds rows (equality, aka probe)

```
SELECT * FROM Person WHERE SSN=543230921
```

Equality query

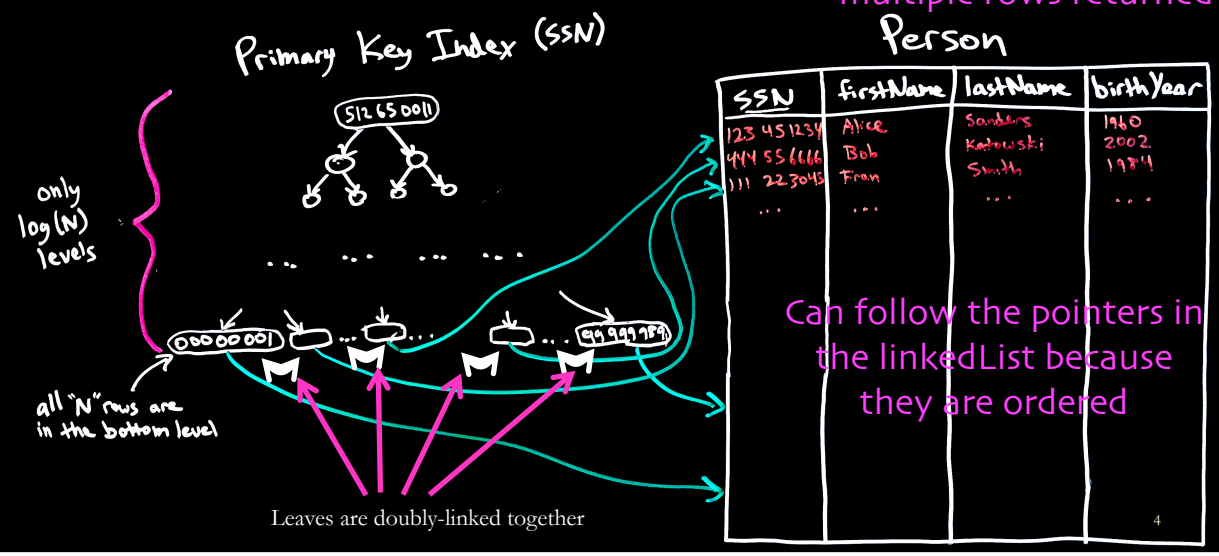
At most Log(n) steps



Review of how an index finds rows (range query)

```
SELECT * FROM Person WHERE SSN>543230921
```

Range Query - multiple rows returned

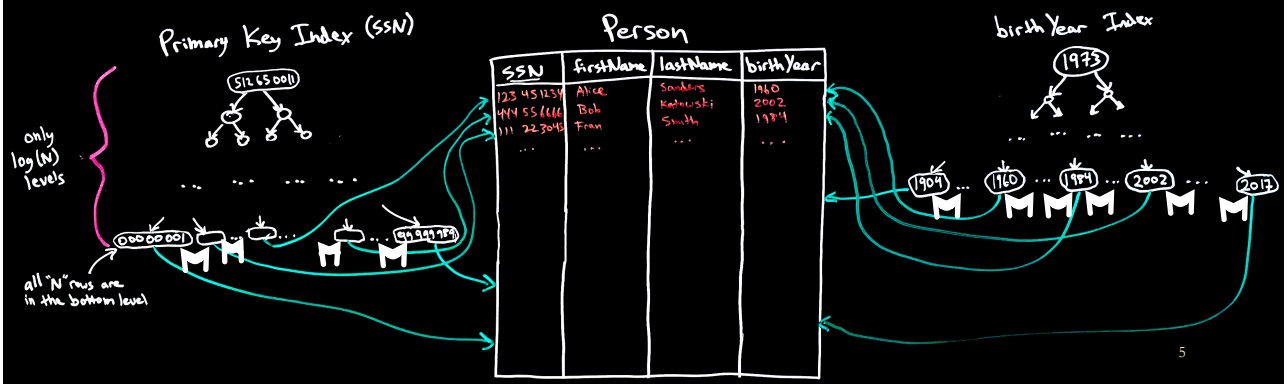


Can follow the pointers in the linkedList because they are ordered

Multiple indexes allow finding rows quickly based on multiple criteria

Query of 2 types

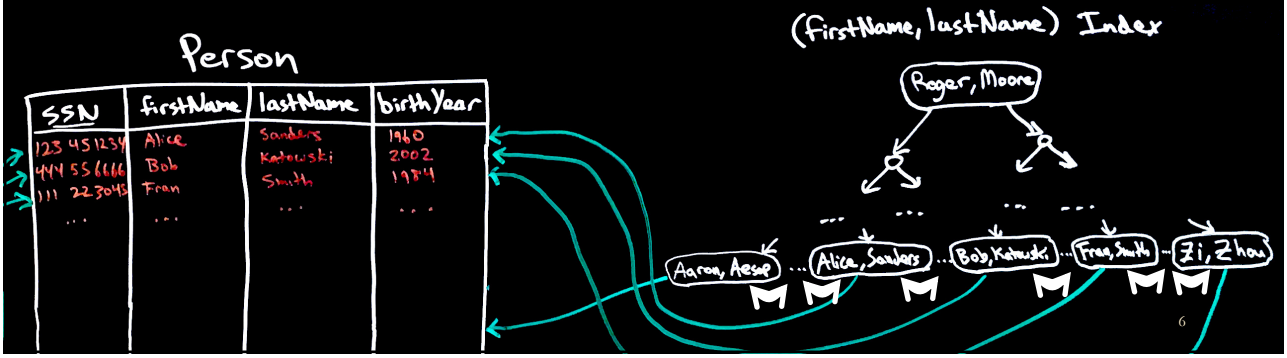
- Need two indexes to quickly get results for both:
  - `SELECT * FROM Person WHERE SSN=543230921`
  - `SELECT * FROM Person WHERE birthYear BETWEEN 1979 AND 1983`



Composite indexes

Sorting index multiple keys

- Useful when WHERE clauses involve pairs of column values
- **SELECT * FROM Person WHERE firstName="Roger" and lastName="Moore"**
- Index sorted in lexicographic order of composite keys
- Requires less space than two separate indexes
- Can find the matching pair of values with one lookup
- Fast if you search for firstName, or "firstName, lastName"
- Slow for lastName alone



B+ Trees in practice (cool facts!)

- A typical B+ tree node has on average 134 children
  - Typical order: 100 (i.e., 100 ≤ keys per node ≤ 200) ?????
  - Typical fill-factor: 67%
  - 2 × 100 × 0.67 = 134
- Index trees rarely have more than 4 or 5 levels:
  - Height 3: 134^3 = 2,406,104 entries
  - Height 4: 134^4 = 322,417,936 entries
  - Height 5: 134^5 = 44,840,334,375 entries
- Top levels can always be in memory:
  - Level 1 = 1 page = 8 KB
  - Level 2 = 134 pages = 1 MB
  - Level 3 = 17,956 pages = 140 MB
  - Level 4 = 2,406,104 pages = 18 GB (OK for a server)
  - Level 5 = 322,417,936 pages = 2.4 TB (probably not in memory)

As you go down in levels, memory storage space required increases rapidly

Key and Index terminology in SQL

- Plain "key" or "index" is just a way to find rows quickly
  - Just create a search tree on that key
- "Unique key" is an index that prevents duplicates ????
  - Bottom level of search tree has no repeated values
  - DBMS can use the tree to quickly search for existing rows with that value before allowing a row insertion (or column update) to proceed
- "Primary key" is just a unique key, but there can only be one per table
  - We think of the primary key as the *most important* unique key in the table
- "Foreign key" makes a column's values match a column in another table
  - The referenced column in the other table should be indexed (usually it's the primary key in the other table)

When to index columns?

- Generally, add an index if the column is:
  - Used in `WHERE` conditions, or
  - Used in `JOIN … ON` conditions, or
  - A foreign key refers to it
- Also helpful if the column is:
  - In a `MIN` or `MAX` aggregation function

(examples follow)

Class quiz 1

- Which indexes would you pick to make the following query fast?

```
SELECT *
FROM employees AS E
WHERE (E.salary > 150000 and E.age=45) OR E.age=30
```

1. Index on <age>
2. Index on <salary>
3. Index on <age, salary>
4. Index on <salary, age>
5. None
6. Something else

<- Correct Answer: 28, 150k 31,50k
because you can speed up both queries (OR)
Leaf Nodes Look like: 50k,31 50k,35

Class quiz 1

- Which indexes would you pick to make the following query fast?

```
SELECT *
FROM employees AS E
WHERE (E.salary > 150000 and E.age=45) OR E.age=30
```

1. Index on <age>
2. Index on <salary>
3. **Index on <age, salary>**
4. Index on <salary, age>
5. None
6. Something else

Class quiz 2

- Which indexes would you pick to make the following query fast?

```
SELECT *
FROM employees AS E
     JOIN departments AS D
       ON E.deptID = D.deptID
WHERE (E.salary > 150000 and E.age=45) OR E.age=30
```

- Index E on <age, salary>
- Index E on <age, salary> and on <deptID> (i.e., 2 indexes)
- Index E on <age, salary, deptID>
- Index D on <deptID>
- No index on D

Class quiz 2

One index per table

- Which indexes would you pick to make the following query fast?

```
SELECT *
FROM employees AS E
     JOIN departments AS D
       ON E.deptID = D.deptID
WHERE (E.salary > 150000 and E.age=45) OR E.age=30
```

- **Index E on <age, salary>**
- Index E on <age, salary> and on <deptID> (i.e., 2 indexes)
- Index E on <age, salary, deptID>
- **Index D on <deptID>**
- No index on D

Class quiz 3

- Which indexes would you pick to make the following query fast?

```
SELECT MAX(E.salary)
FROM employees AS E
WHERE E.age=30
```

1. Index on <age>
2. Index on <salary>
3. Index on <age, salary>
4. Index on <salary, age>
5. Index on <age> and on <salary>
6. None
7. Something else

<- this way you don't need to traverse the entire list, start at right side end

Class quiz 3

- Which indexes would you pick to make the following query fast?

```
SELECT MAX(E.salary)
FROM employees AS E
WHERE E.age=30
```

1. Index on <age>
2. Index on <salary>
3. **Index on <age, salary>**
4. Index on <salary, age>
5. Index on <age> and on <salary>
6. None
7. Something else