MLDS-413 Introduction to Databases and Information Retrieval

Lecture 15
Indexing Databases

Instructor: Nikos Hardavellas

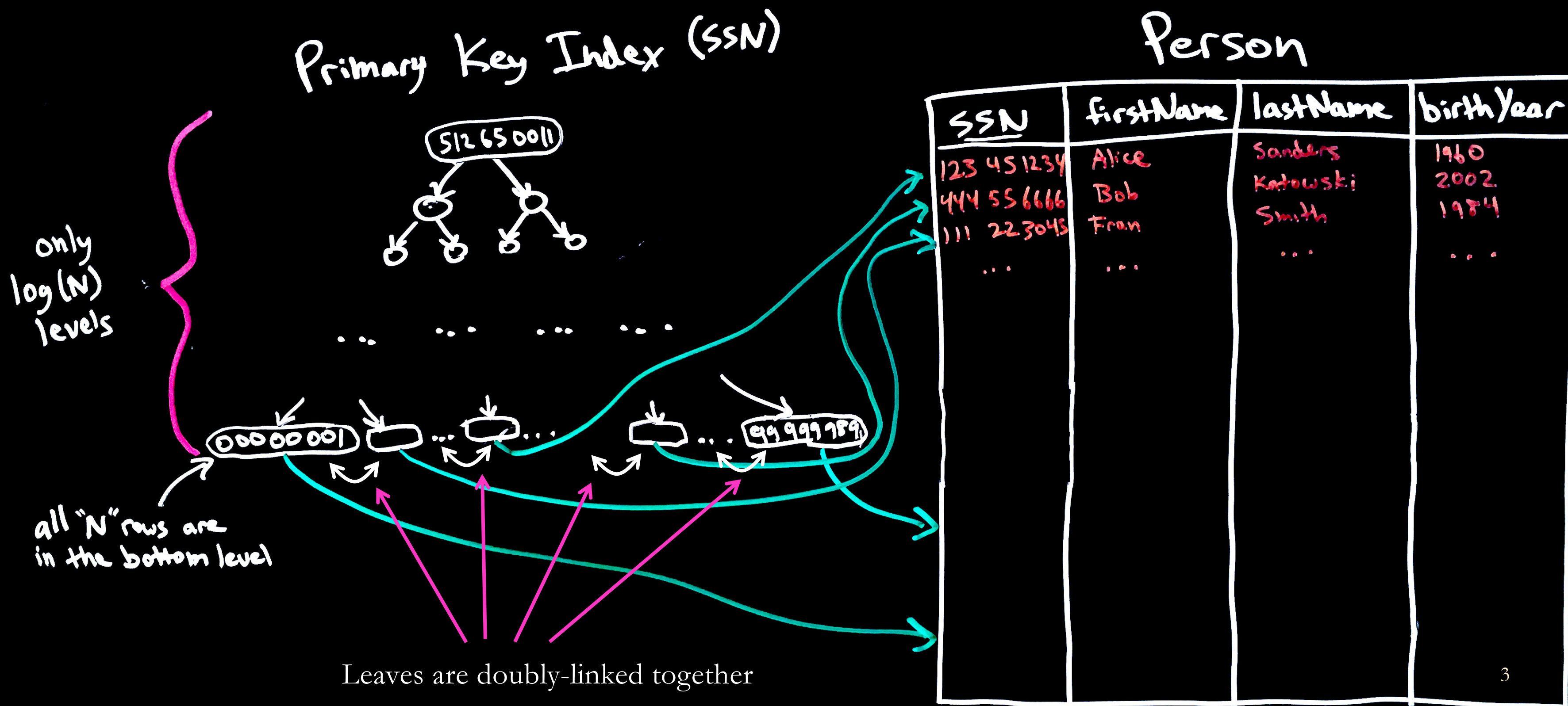
Slides adapted from Steve Tarzia

Last Lecture

- 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 remain 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)

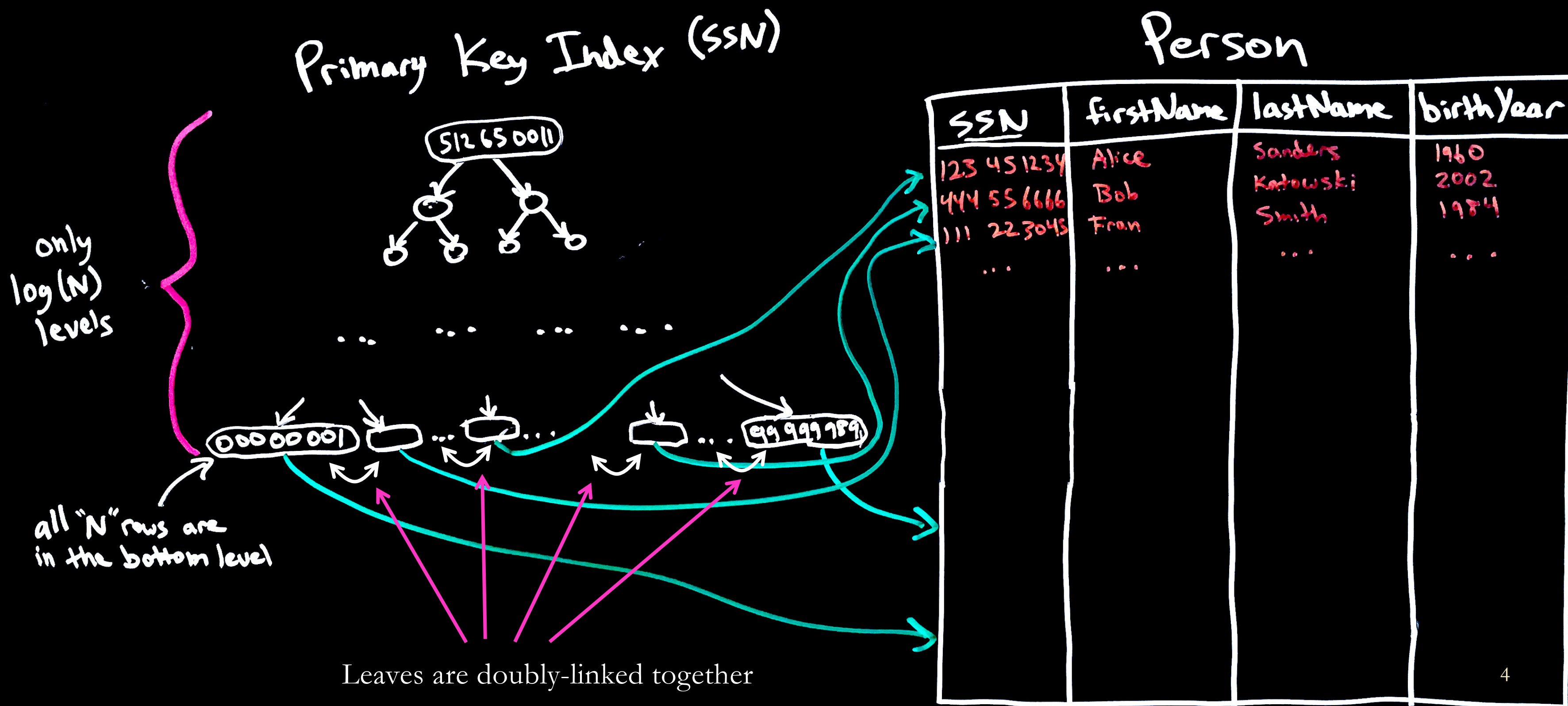
Review of how an index finds rows (equality, aka probe)

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



Review of how an index finds rows (range query)

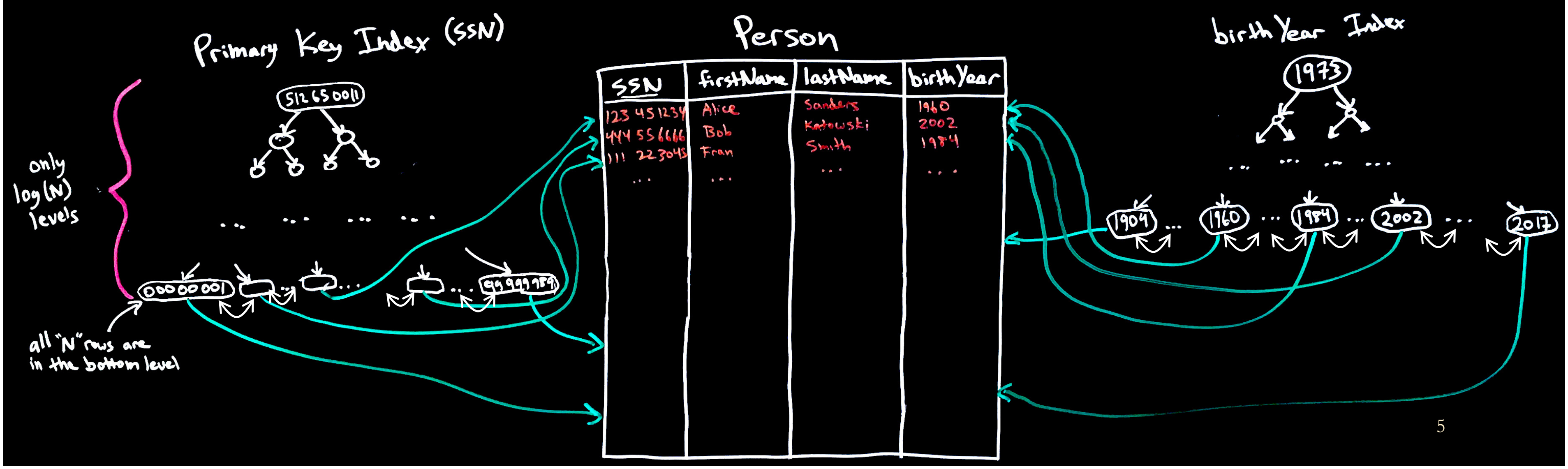
```
SELECT * FROM Person WHERE SSN>543230921
```



Leaves are doubly-linked together

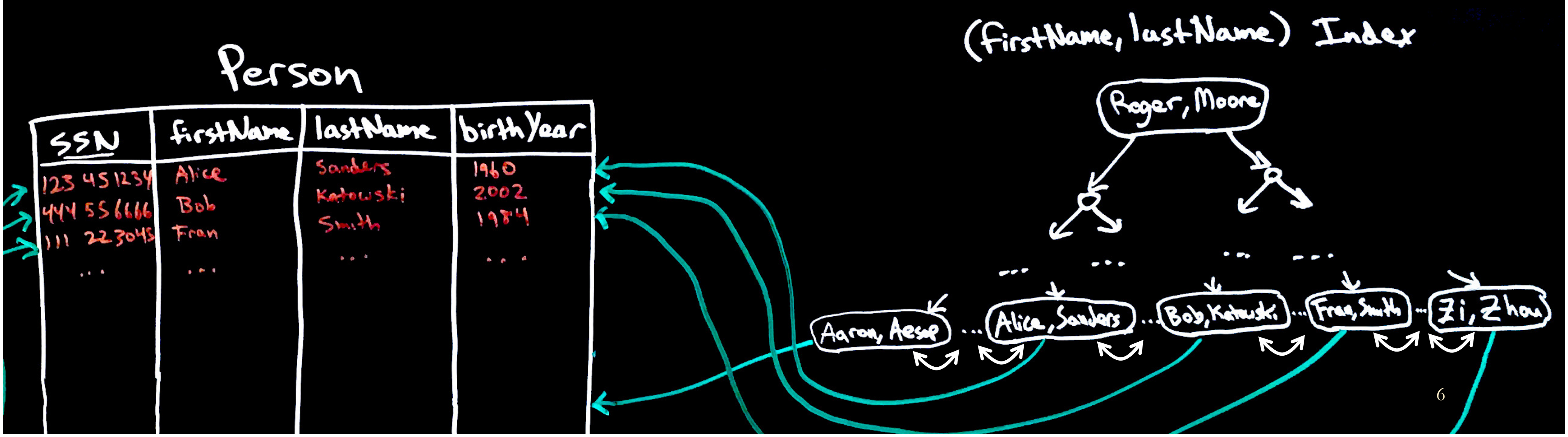
Multiple indexes allow finding rows quickly based on multiple criteria

- 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

- 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, e.g., index on <firstName, lastName>
- Requires less space than two separate indexes
- Can find the matching pair of values with one lookup
- The index above is fast if you search for firstName, or "firstName, lastName"
- ...but 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%
  - $2 \times 100 \times 0.67 = 134$
- 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)

Key and Index terminology in SQL

- Plain "key" is just a field (or combination of) we can use to find rows quickly
- An "index" on that "key" is just a data structure we can use to find rows quickly
  - Just create a search tree (B+ tree) on that key
- "Unique key" is a key that prevents duplicates
  - Bottom level of index (B+ tree) on the unique key 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
  - 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

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
```

1. Index E on <age, salary>
2. Index E on <age, salary> and on <deptID> (i.e., 2 indexes on table E)
3. Index E on <age, salary, deptID>
4. Index D on <deptID>
5. No index on D

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
```

1. **Index E on <age, salary>**
2. Index E on <age, salary> and on <deptID> (i.e., 2 indexes on table E)
3. Index E on <age, salary, deptID>
4. **Index D on <deptID>**
5. 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

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

Creating indexes

- Indexes are usually defined when the table is created
  - Usually define at least a *primary key*
- But you may later realize that certain queries are too slow
  - Without proper indexes, DBMS will have to examine every row in the table to find the relevant rows
  - Adding one or more indexes may dramatically speed up a query

Basic syntax:

```
CREATE INDEX index_name ON table_name (column_name)
```