Indexing 2: hash indexes, join indexes, and models of indexability

Lecture 4
2ID35, Spring 2015

George Fletcher

Faculteit Wiskunde & Informatica
Technische Universiteit Eindhoven

1 May 2015

Where we've been

Last time

- basics of indexing

Where we've been

Last time

- basics of indexing
  - clustering vs. nonclustering
  - dense vs. sparse
  - ordered vs. hashed

Where we've been

Last time

- basics of indexing
  - clustering vs. nonclustering
  - dense vs. sparse
  - ordered vs. hashed
- ordered indexes: B+ trees
  - supports range search, with only logarithmic overhead

Where we've been

Last time

- basics of indexing
  - clustering vs. nonclustering
  - dense vs. sparse
  - ordered vs. hashed
- ordered indexes: B+ trees
  - supports range search, with only logarithmic overhead
- ordered indexes: R trees
  - spatial search, with only logarithmic overhead

Where we've been

Last time

- basics of indexing
  - clustering vs. nonclustering
  - dense vs. sparse
  - ordered vs. hashed
- ordered indexes: B+ trees
  - supports range search, with only logarithmic overhead
- ordered indexes: R trees
  - spatial search, with only logarithmic overhead
- ordered indexes: GiST
  - extensible generalized search-tree infrastructure

Where we're going

Today's agenda

- Indexing, part 2
  - hash-based indexing

Where we're going

Today's agenda

- Indexing, part 2
  - hash-based indexing
  - join indexing

Where we're going

Today's agenda

- Indexing, part 2
  - hash-based indexing
  - join indexing
  - models of indexability

Indexing

Two basic types of indexes

- Ordered indexes. based on a sorted ordering of the key values
- Hash indexes. based on a uniform distribution of key values across a finite number B of buckets. The bucket to which a value is assigned is determined by a hash function.
  - i.e., a function h which assigns each element k of the key space to an integer $h(k) \in \{0, \ldots, B-1\}$

Indexing

Two basic types of indexes

- Ordered indexes. based on a sorted ordering of the key values
- Hash indexes. based on a uniform distribution of key values across a finite number B of buckets. The bucket to which a value is assigned is determined by a hash function.
  - i.e., a function h which assigns each element k of the key space to an integer $h(k) \in \{0, \ldots, B-1\}$
  - e.g., for strings, the sum over integer representations of each character, modulo B

Static hashing

Static hashing: hash file

A-217	Brighton	750
A-101	Downtown	500
A-110	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

Static hashing: hash file

A-217	Brighton	750
A-101	Downtown	500
A-110	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

bucket 0

bucket 1

bucket 2

bucket 3

A-217	Brighton	750
A-305	Round Hill	350

bucket 4

A-222	Redwood	700

bucket 5

A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700

bucket 6

bucket 7

A-215	Mianus	700

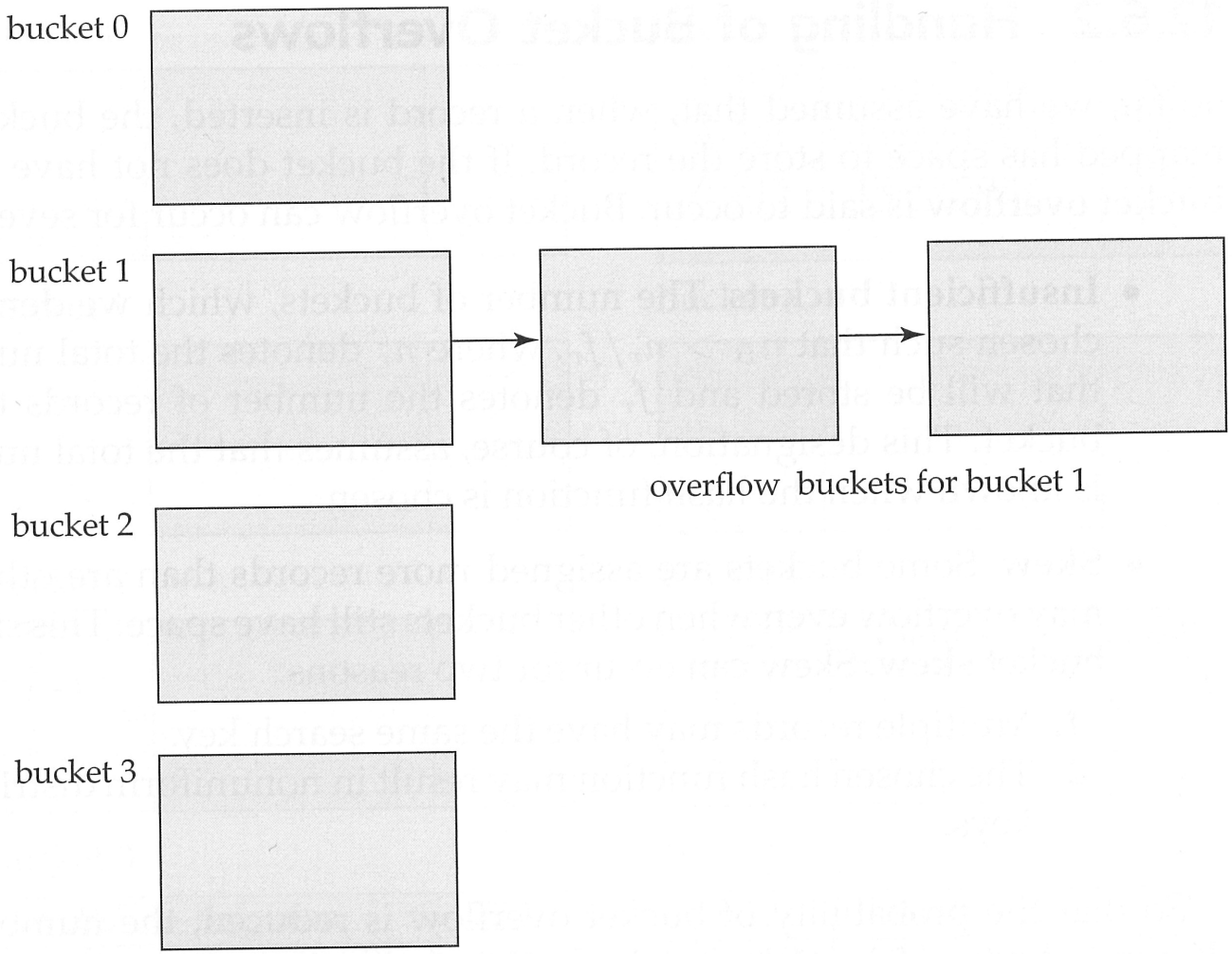
bucket 8

A-101	Downtown	500
A-110	Downtown	600

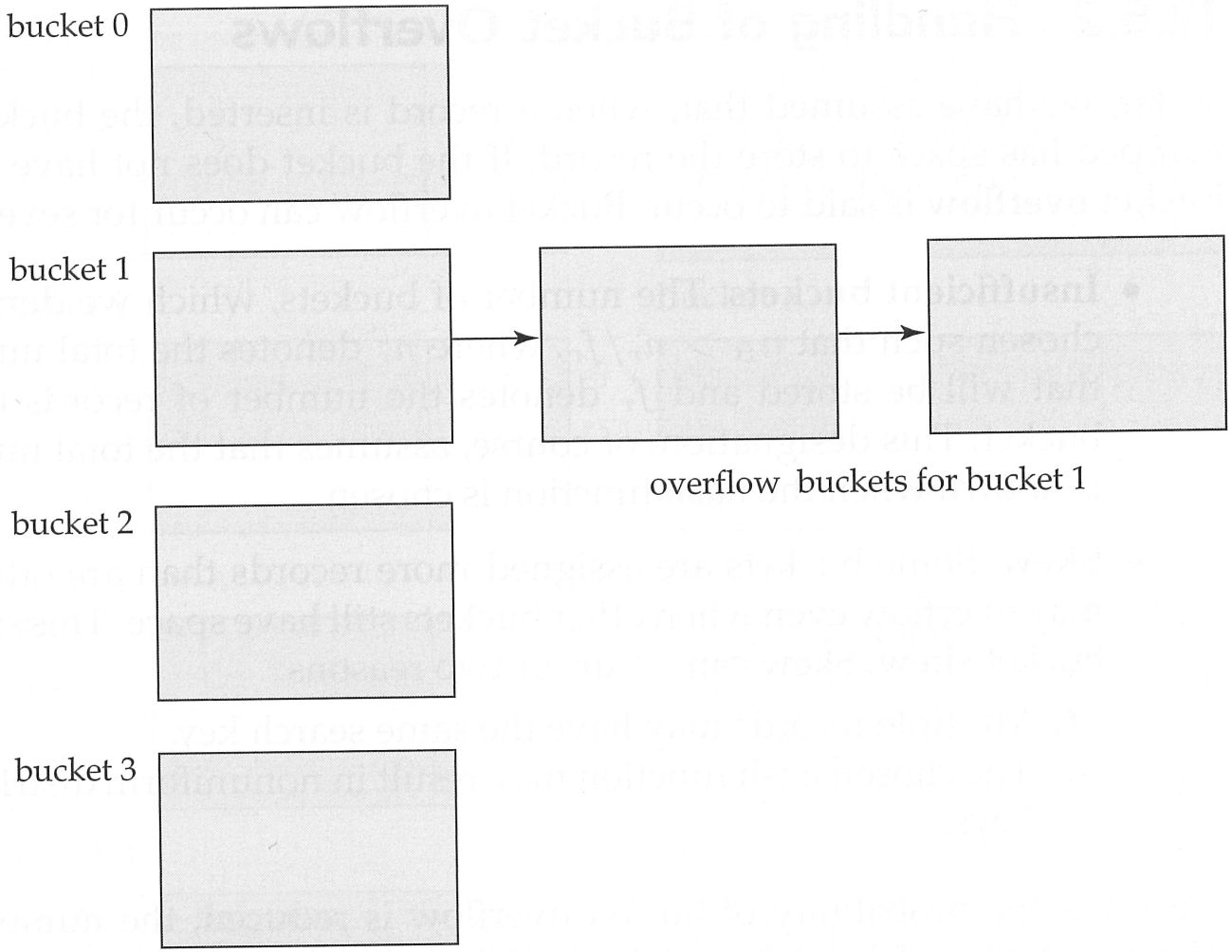
bucket 9

with branch name as key

Static hashing: overflow chaining (closed hashing)

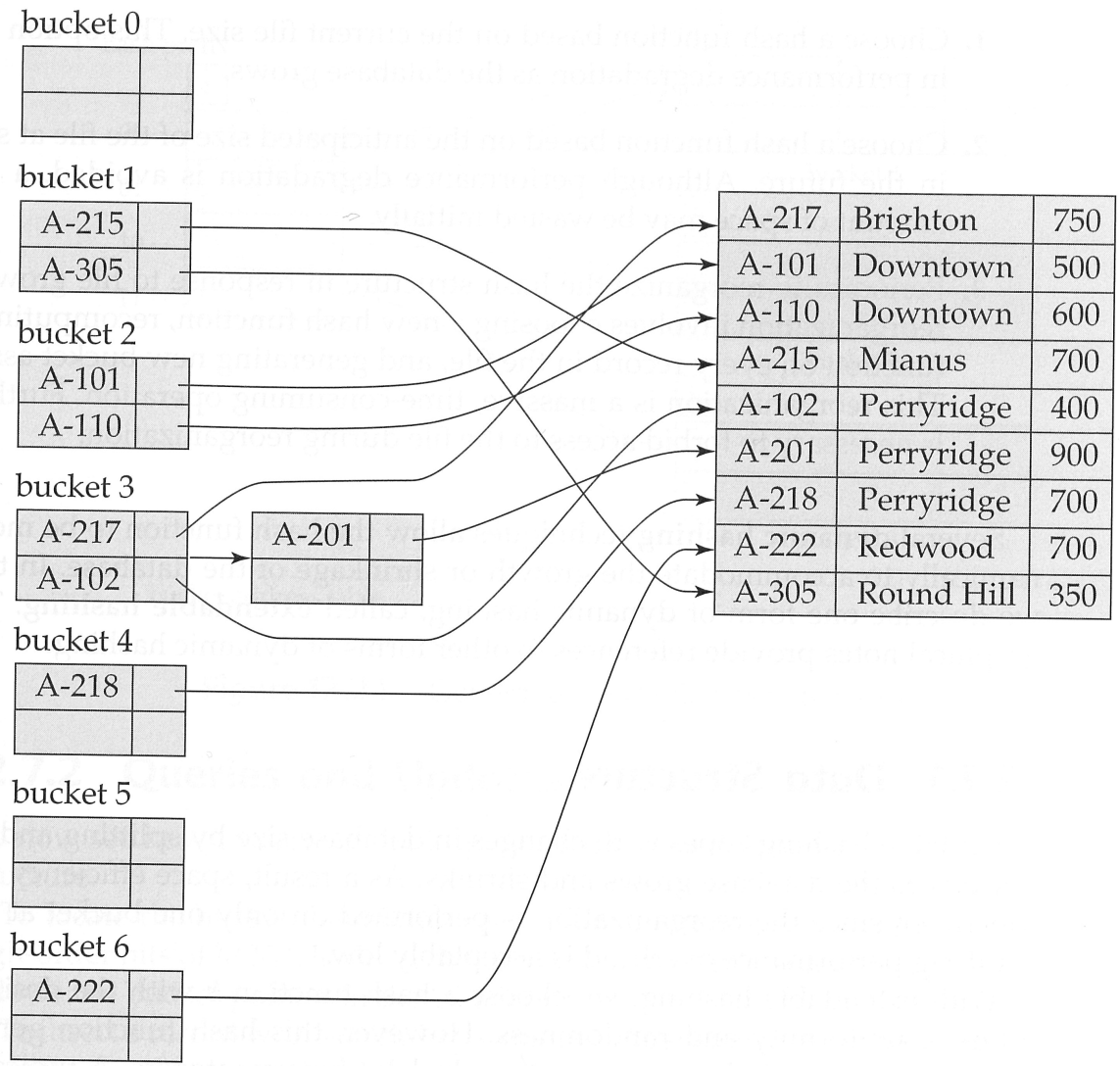


Static hashing: overflow chaining (closed hashing)



vs. open hashing (e.g., linear probing)

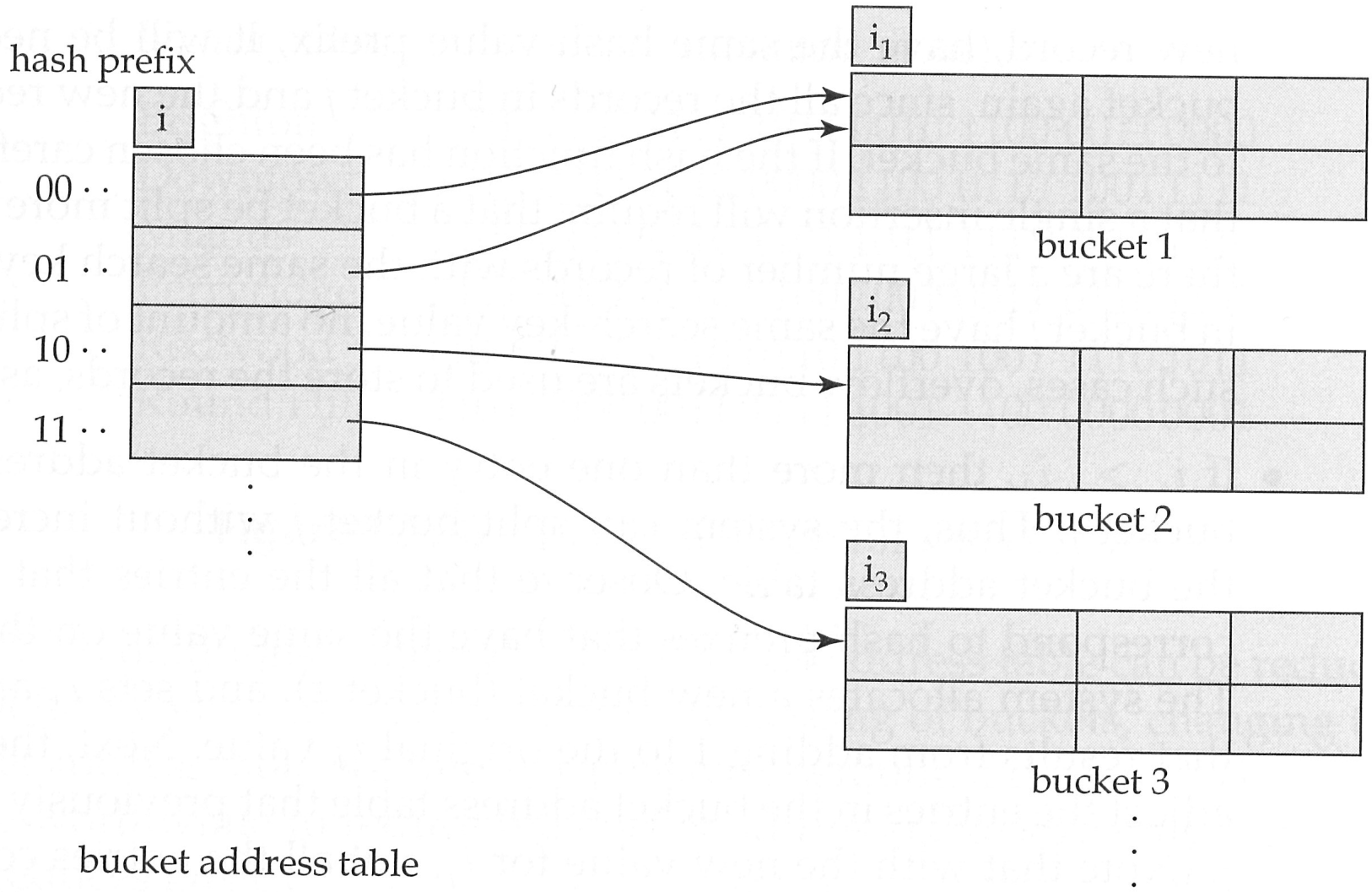
Static hashing: hash-based indexing



secondary index on account number (hash value: sum of digits modulo 7)

Dynamic hashing

Dynamic hashing: extendable hash index



Dynamic hashing: extendable hash index

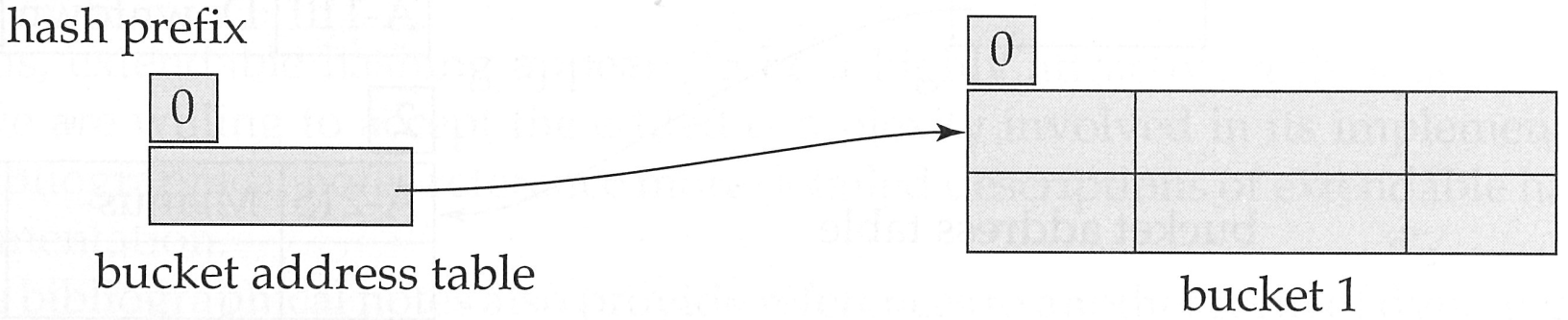
A-217	Brighton	750
A-101	Downtown	500
A-110	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

Dynamic hashing: extendable hash index

A-217	Brighton	750
A-101	Downtown	500
A-110	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

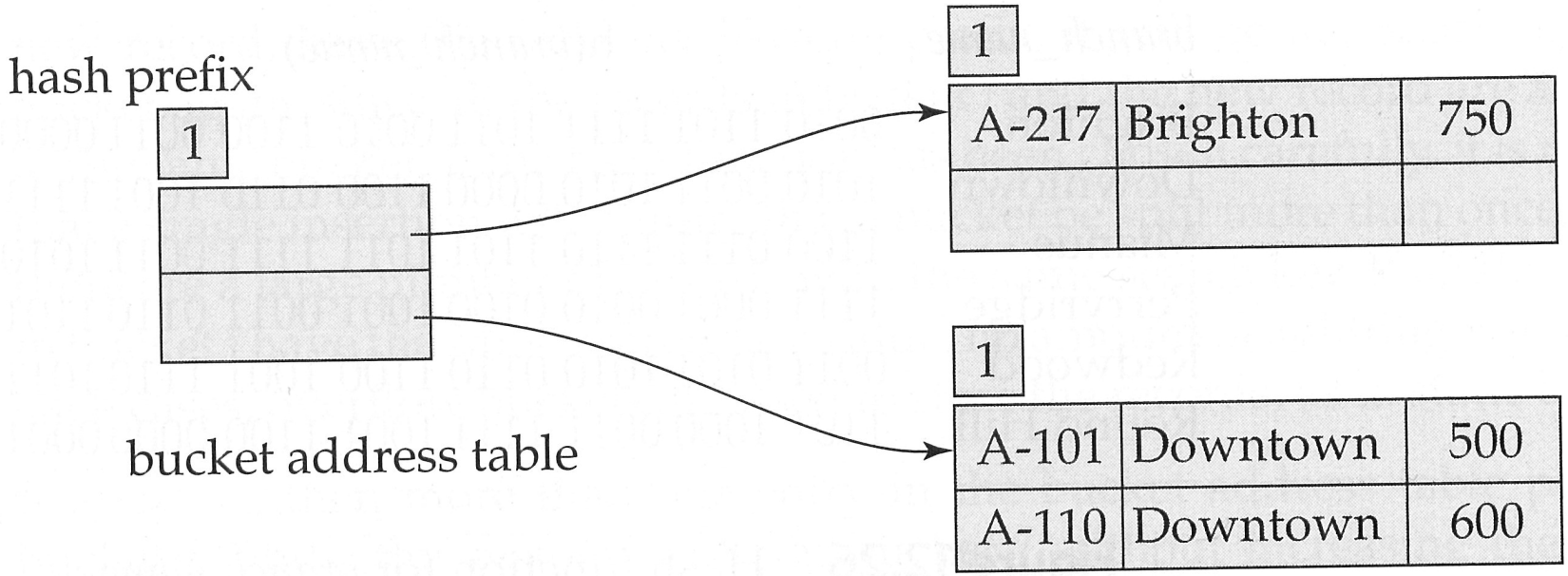
branch_name	h(*branch_name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001

Dynamic hashing: extendable hash index



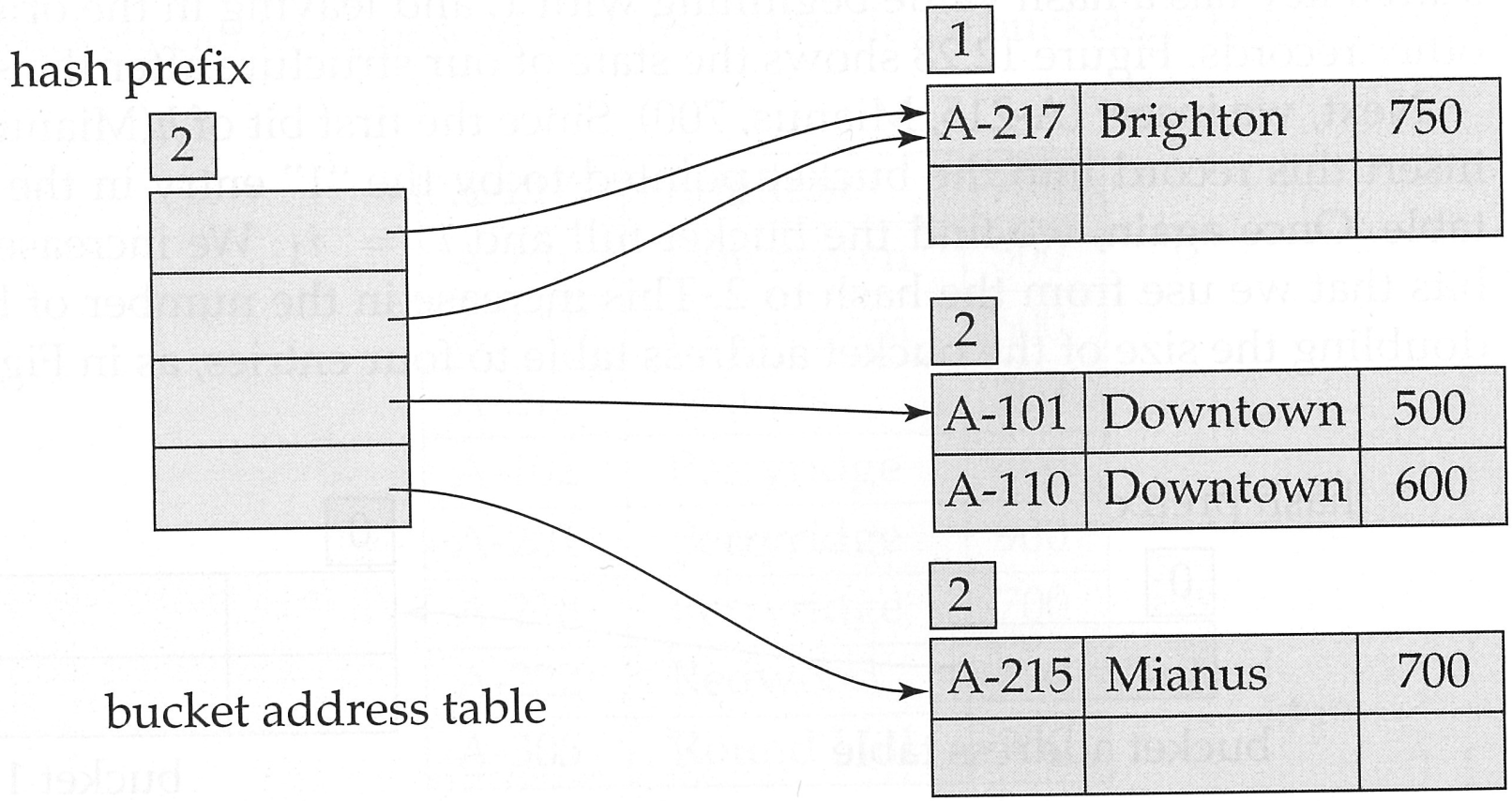
initial hash index

Dynamic hashing: extendable hash index



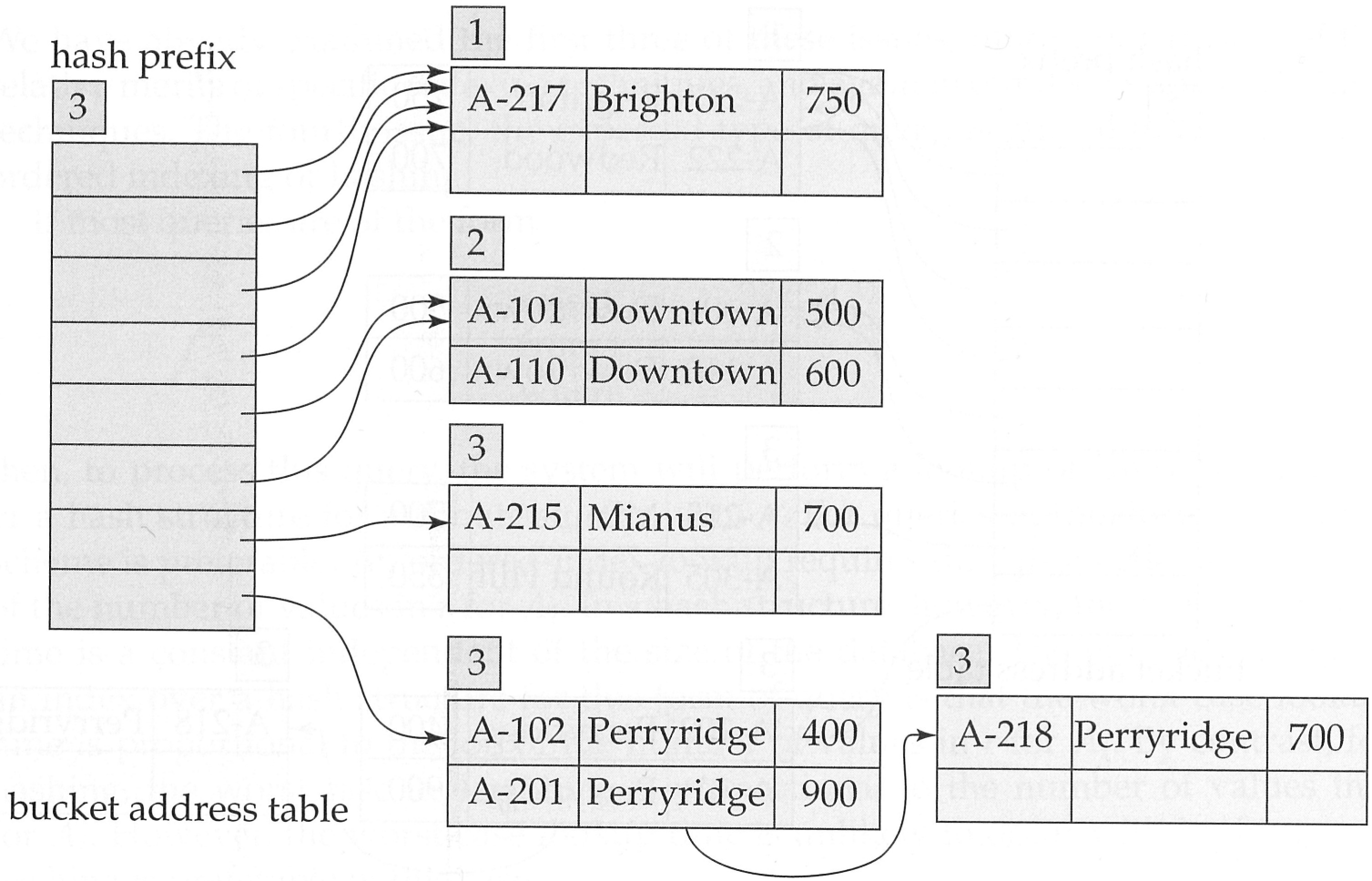
after insertion of:
(A-217, Brighton, 750):0010,
(A-101, Downtown, 500):1010,
(A-110, Downtown, 600):1010

Dynamic hashing: extendable hash index



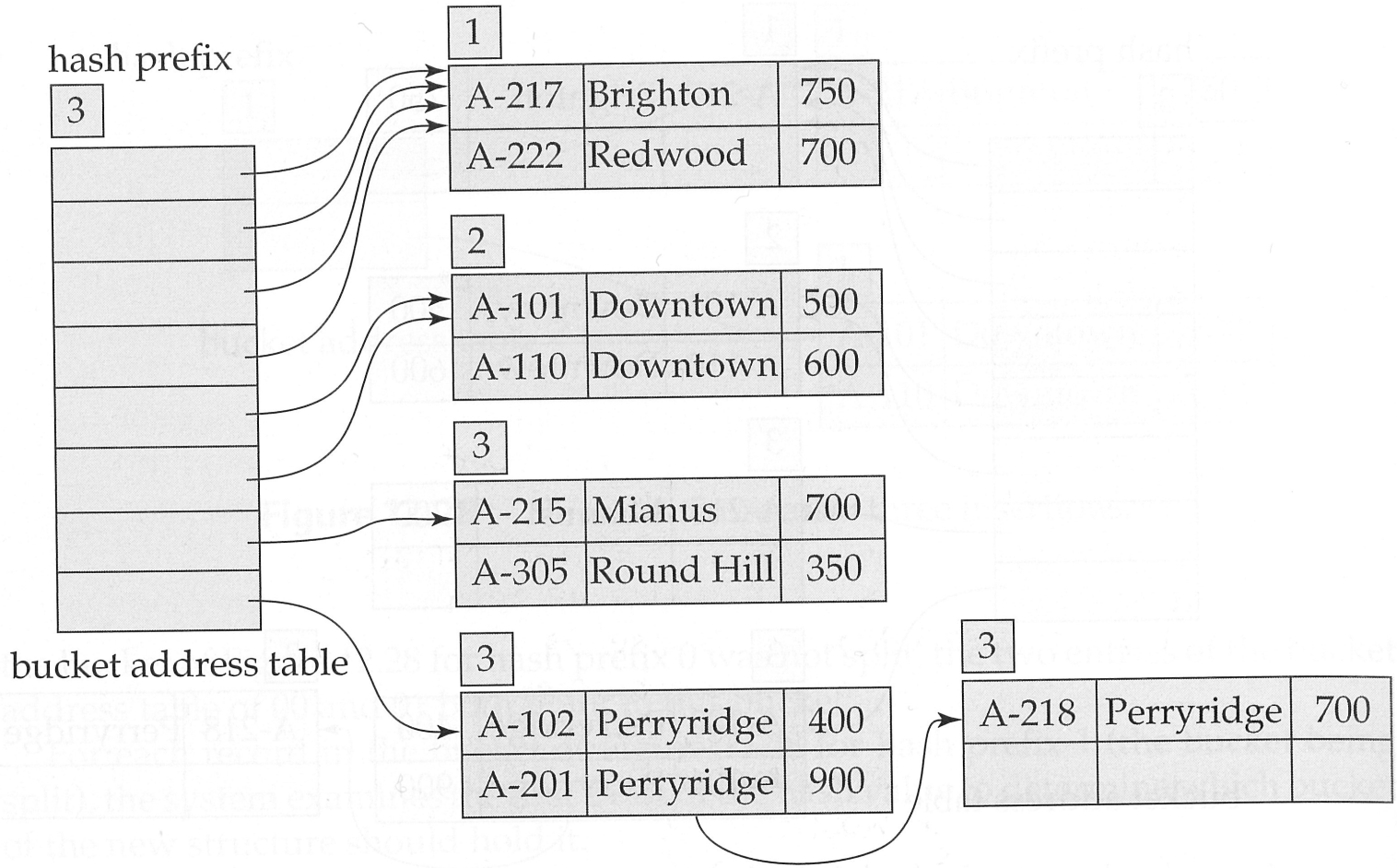
after insertion of:
(A-215, Mianus, 700):1100

Dynamic hashing: extendable hash index



after insertion of: (A-102, Perryridge, 400):1111, (A-201, Perryridge, 900):1111, (A-218, Perryridge, 700):1111

Dynamic hashing: extendable hash index



after insertion of: (A-222, Redwood, 700):0011,
(A-305, Round Hill, 350):1101

Dynamic hashing: extendable hash index

Deletions

- essentially the reverse of insertions
- merge locally when buckets are empty
- shrink bucket address table when all local hash prefixes fall below the global hash prefix

Dynamic hashing: extendable hash index

- extendable hashing requires two or three I/Os per search
- shortcomings of extendable hashing
  - extra overhead of directory, both for storage and look-up

Dynamic hashing: extendable hash index

- extendable hashing requires two or three I/Os per search
- shortcomings of extendable hashing
  - extra overhead of directory, both for storage and look-up
- can we eliminate the directory?

Dynamic hashing: extendable hash index

- extendable hashing requires two or three I/Os per search
- shortcomings of extendable hashing
  - extra overhead of directory, both for storage and look-up
- can we eliminate the directory?
- yes, use a family of hash functions to simulate the directory
  - linear hash index

Dynamic hashing: linear hash index

Idea:

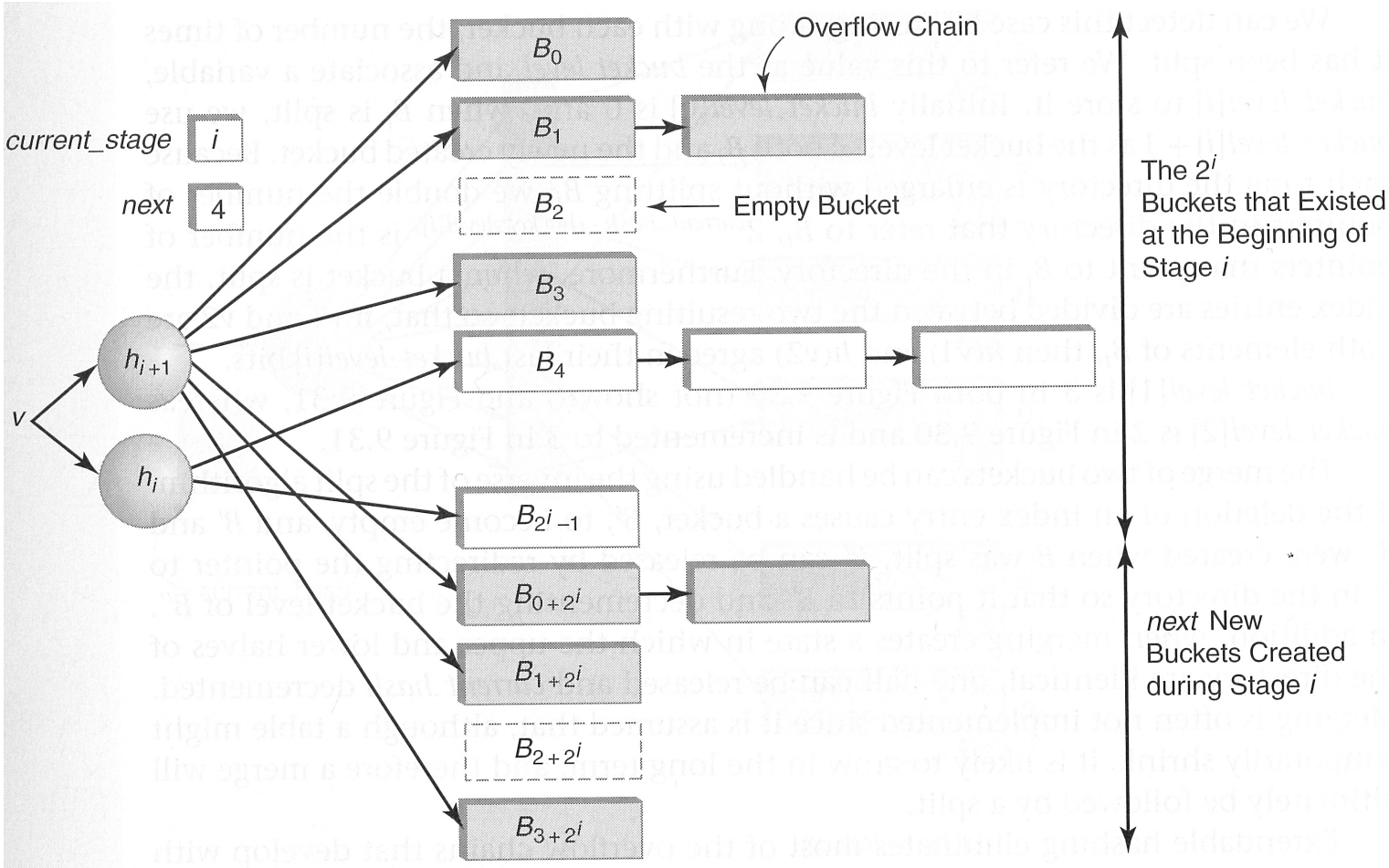
- use a family of hash functions $h_0, \ldots, h_n$, where h_i uses i (least significant) bits to determine the appropriate bucket
- at stage i, there are initially 2^i buckets, and we use h_i for search

Dynamic hashing: linear hash index

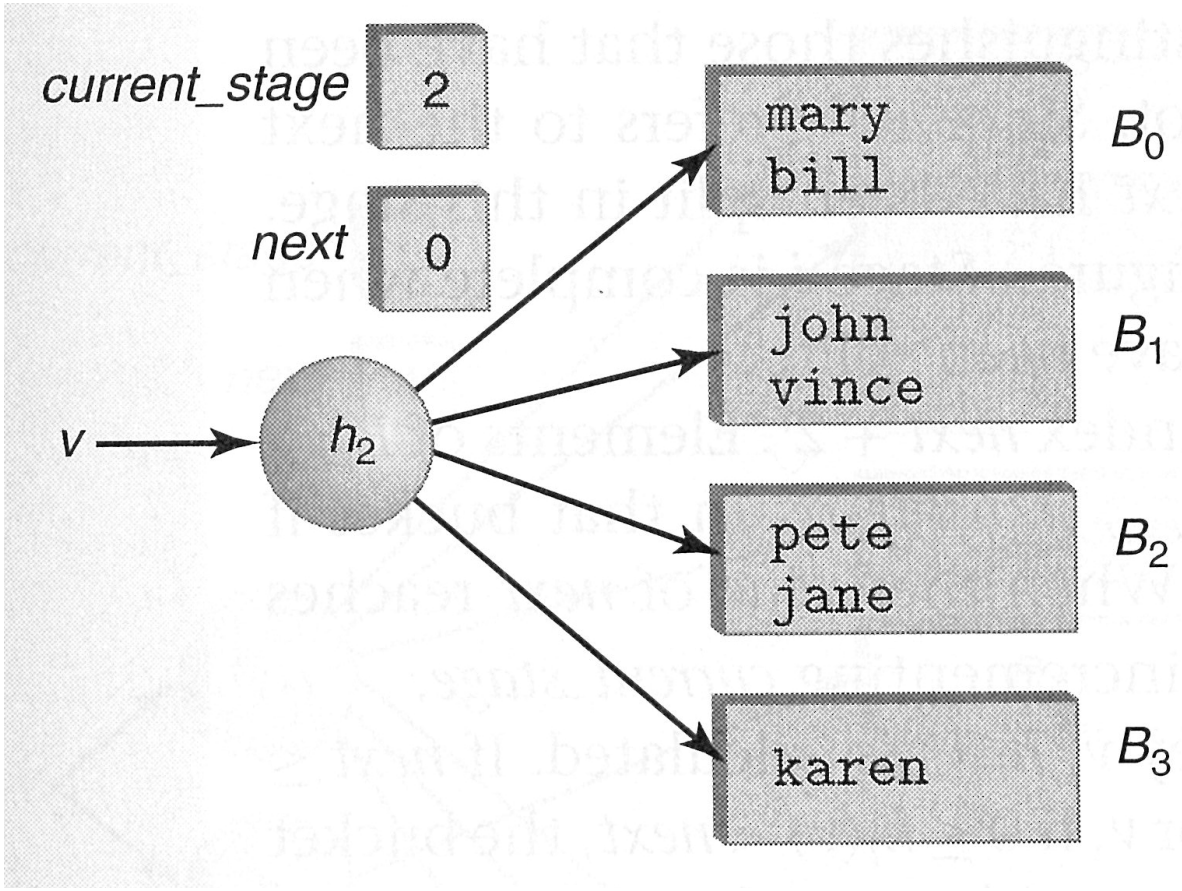
Idea:

- use a family of hash functions $h_0, \ldots, h_n$, where h_i uses i (least significant) bits to determine the appropriate bucket
- at stage i, there are initially 2^i buckets, and we use h_i for search
- to grow, systematically split and rehash buckets from 0 to 2^i, as needed
  - keep pointer to *next* bucket to split
  - split upon overflow (of any bucket)
  - after split, $next = next + 1$
- for those buckets which have already been split, use h_{i+1} for search

Dynamic hashing: linear hash index

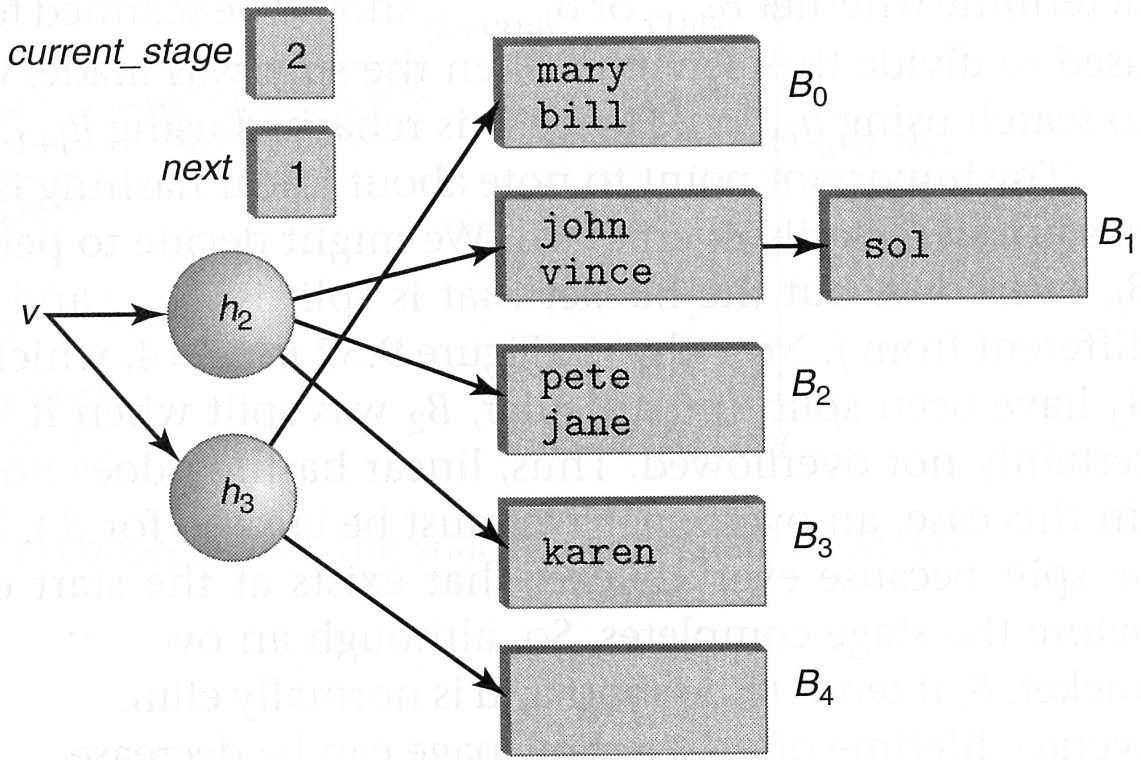


Dynamic hashing: linear hash index



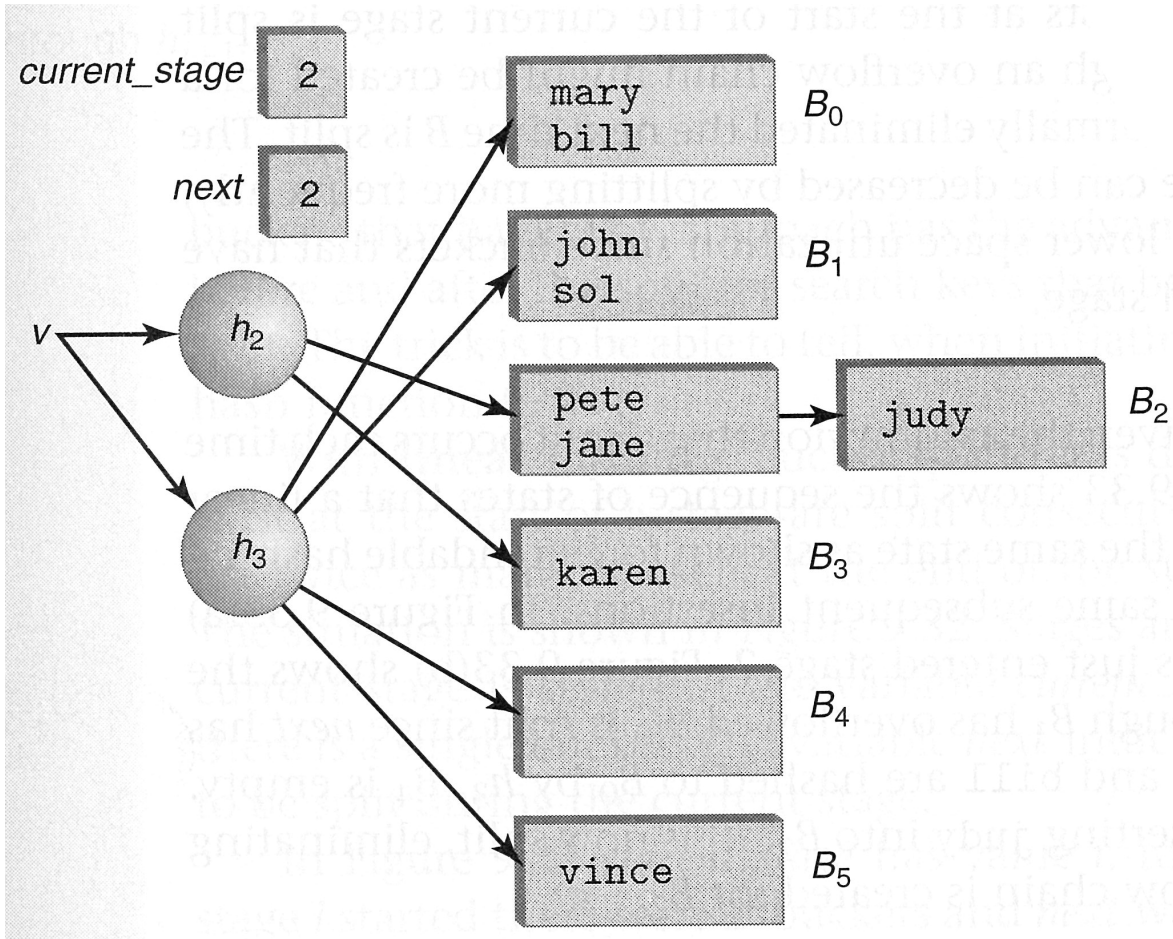
initial table at stage 2

Dynamic hashing: linear hash index



after insertion of "sol"

Dynamic hashing: linear hash index



after insertion of "judy"

Exercise: linear hash index

A-217	Brighton	750
A-101	Downtown	500
A-110	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

branch_name	h(*branch_name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001

Suppose L is an empty linear hash table, where each page holds two records.
Show L after inserting all account records (hashed on branch name, as in the examples above), in order of their appearance.

Ordered vs. Hash-based Indexing

Ordered vs. hash-based index

- space overhead
- supported query types
- I/O costs
  - search
  - maintenance

Bitmap indexes

Bitmap indexes

Bitmap index

- implemented as one or more bit vectors
- ideal for selections on attributes of small domain cardinality
  - job category { entry level, lead, management }
  - academic degree { BSc, MSc PhD }
  - sex { female, male }
  - smoker { yes, no }
- the *i*th bit in the "male" vector is 1 if, in the *i*th row of the Person table, the Sex attribute has value Male.

Bitmap indexes

record number	*name*	*gender*	*address*	*income_level*
0	John	m	Perryridge	L1
1	Diana	f	Brooklyn	L2
2	Mary	f	Jonestown	L1
3	Peter	m	Brooklyn	L4
4	Kathy	f	Perryridge	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Bitmap indexes

Bitmap index

- only one bit per row, per domain value
- bitmaps are very small, compared to actual relation size, usually less than 1%
- also, highly compressible

Bitmap indexes

Bitmap index

- only one bit per row, per domain value
- bitmaps are very small, compared to actual relation size, usually less than 1%
- also, highly compressible
- complex boolean selection conditions over bitmaps can be performed efficiently using bitwise AND, OR, NOT
- very successful in read-intensive applications in data mining and data warehousing

Join indexes

Join indexes

CUSTOMER

csur	cname	city	age	job
1	Smith	Boston	21	clerk
2	Collins	Austin	26	secretary
3	Ross	Austin	36	manager
4	Jones	Paris	29	engineer

CP

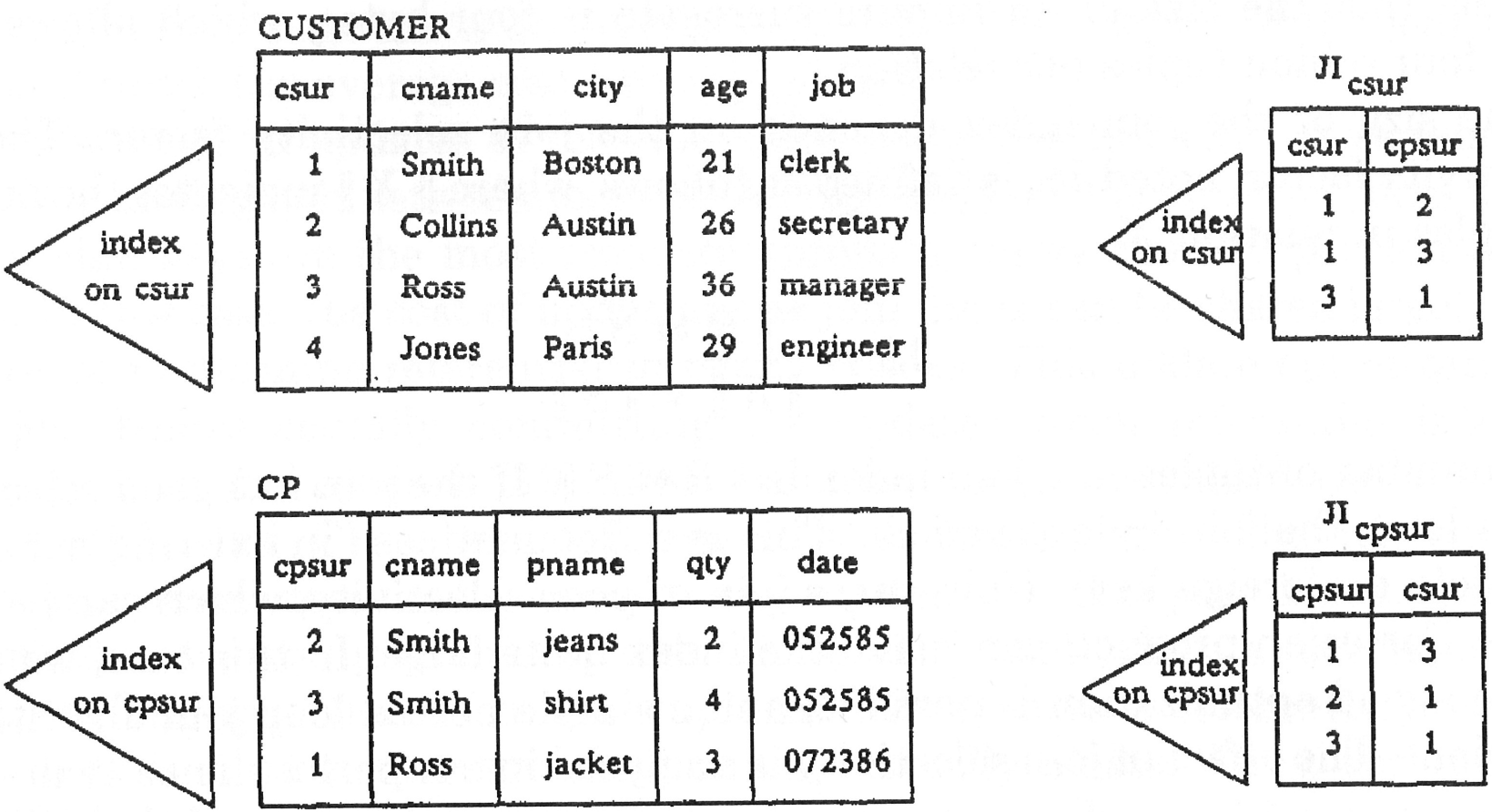
cpsur	cname	pname	qty	date
2	Smith	jeans	2	052585
3	Smith	shirt	4	052585
1	Ross	jacket	3	072386

JI

csur	cpsur
1	2
1	3
3	1

JI on attribute cname

Join indexes: implementation



indexing on surrogates

Join indexes: implementation

can also be implemented as "bitmapped" join index

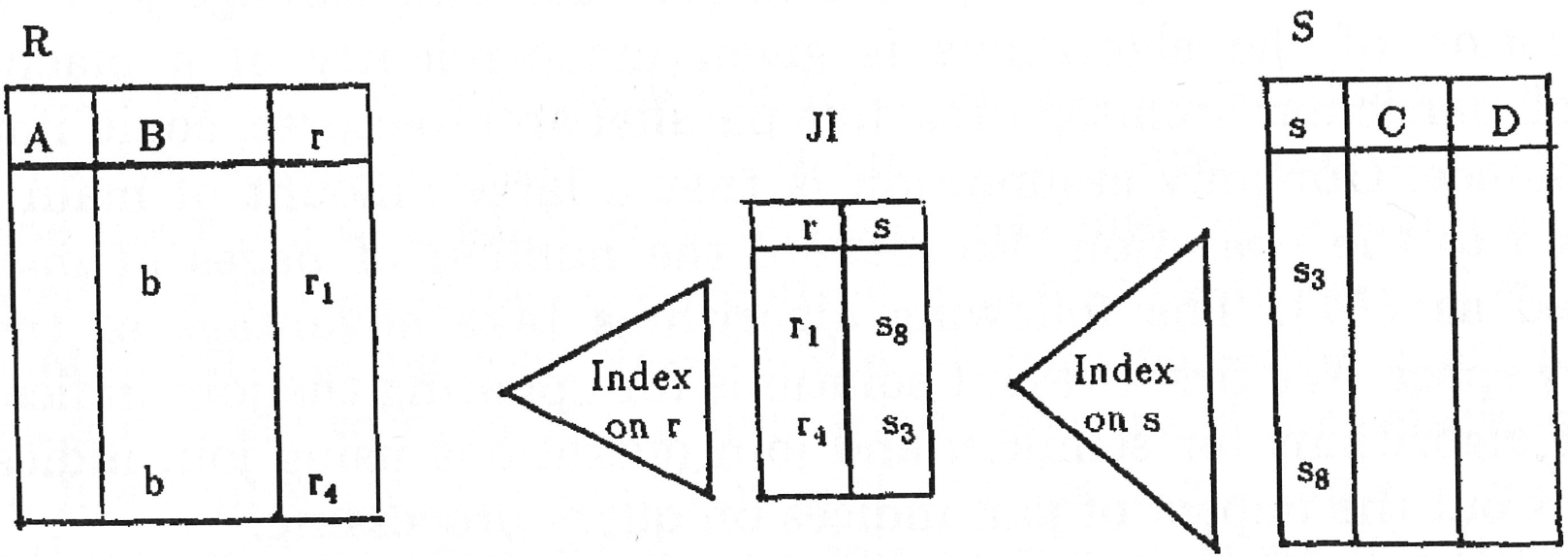
- instead of

$$(rid, sid)$$

we have

$$(rid, \text{bitmap for matching tuples in } S)$$

Join indexes: use



R.B = ‘b’ and R.A=S.D

Join indexes: use

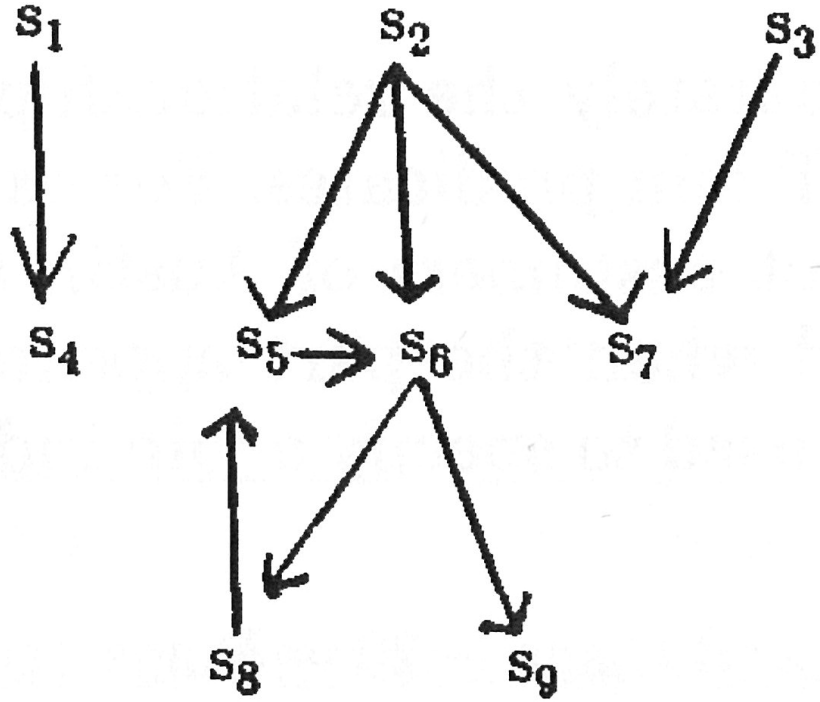
	Ph.D				JI	
sur	advisee	advisor	university	year	sur	sur
1	Doe		Harvard	1970	1	2
2	Smith	Doe	Harvard	1976	2	3
3	Ross	Smith	MIT	1980	4	5
4	Hayes		Stanford	1978		
5	James	Hayes	Princeton	1984		

coding a self-join

Join indexes: use

JI

s_1	s_4
s_2	s_5
s_2	s_6
s_2	s_7
s_3	s_7
s_5	s_6
s_6	s_8
s_6	s_9
s_8	s_5



coding a directed graph

Join indexes: use

Many variations in practice, e.g., extensions for OO hierarchies

Models of indexing

Indexing

- key: a collection of attributes (of a relation)

Indexing

- key: a collection of attributes (of a relation)
- index: data structure for external searching
  - i.e., mapping key values to data
  - i.e., locating a collection of data entries $k*$ for search key k
  - i.e.,

    value $\mapsto$ index
    $\mapsto$ blocks holding records
    $\mapsto$ matching records

Indexing

- key: a collection of attributes (of a relation)
- index: data structure for external searching
  - i.e., mapping key values to data
  - i.e., locating a collection of data entries $k*$ for search key k
  - i.e.,
    value $\mapsto$ index
    $\mapsto$ blocks holding records
    $\mapsto$ matching records
- basic operations: search(k), insert(k), delete(k)

Models of Indexing

1. workload model (Hellerstein et al)
2. GMAP model (Tsatalos et al)
3. query language model (Fletcher et al)

Models of Indexing: workload model

(1) Workload model

- proposed and studied by Hellerstein et al, *JACM* 2002
- studies indexing schemes with respect to "workloads" of queries

Models of Indexing: workload model

A workload is modeled as a finite subset of some domain, together with a set of queries

$$W = (D, I, \mathcal{Q})$$

where

- D is a domain (e.g., $\mathbb{R}$ with $\leq$)

Models of Indexing: workload model

A workload is modeled as a finite subset of some domain, together with a set of queries

$$W = (D, I, \mathcal{Q})$$

where

- D is a domain (e.g., $\mathbb{R}$ with $\leq$)
- I is a finite subset of D, called the instance

Models of Indexing: workload model

A workload is modeled as a finite subset of some domain, together with a set of queries

$$W = (D, I, \mathcal{Q})$$

where

- D is a domain (e.g., $\mathbb{R}$ with $\leq$)
- I is a finite subset of D, called the instance
- $\mathcal{Q} = \{Q_1, \ldots, Q_q\}$ is a finite set of queries, which are

  $$Q_i \subseteq I$$

  for each $1 \leq i \leq q$

Models of Indexing: workload model

A workload is modeled as a finite subset of some domain, together with a set of queries

$$W = (D, I, \mathcal{Q})$$

For example, the one-dimensional range queries

- $D = \mathbb{N}$, with natural ordering $\leq$
- $I = \{i \mid 1 \leq i \leq n\}$, for some fixed $n \in \mathbb{N}$
- $\mathcal{Q} = \{Q[a, b] \mid 1 \leq a \leq b \leq n\}$, where $Q[a, b] = \{i \mid a \leq i \leq b\}$

Models of Indexing: workload model

An indexing scheme $\mathcal{S}$ for workload $W = (D, I, \mathcal{Q})$ is a collection of blocks $b_1, \ldots, b_k$ where each block is a B sized subset of I, such that $\bigcup b_i = I$, for some fixed positive integer B.

- in practice, B is on the order of 100 to 1,000

Models of Indexing: workload model

Two performance measures: storage and access cost

- Storage redundancy of $\mathcal{S}$ is the maximum number of blocks that contain an element of I.

Models of Indexing: workload model

Two performance measures: storage and access cost

- Storage redundancy of $\mathcal{S}$ is the maximum number of blocks that contain an element of I.
- The average redundancy is then the average number of blocks that contain an element of I, i.e., $\frac{kB}{|I|}$
  - in the range of 1 (best) to k (worst)
    - best when $|I| = kB$
    - worst when $|I| = B$

Models of Indexing: workload model

Two performance measures: storage and access cost

- Given query $Q \in \mathcal{Q}$, the ideal access cost would be

$$\left\lceil \frac{|Q|}{B} \right\rceil$$

blocks. If we let C_Q be a minimum-sized set of blocks which actually covers Q, then the access overhead of Q in $\mathcal{S}$ is

$$A(Q) = \frac{|C_Q|}{\left\lceil \frac{|Q|}{B} \right\rceil}$$

Models of Indexing: workload model

Then, the access overhead of $\mathcal{S}$ is

$$A(\mathcal{S}) = \max_{Q \in \mathcal{Q}} A(Q)$$

Models of Indexing: workload model

Then, the access overhead of $\mathcal{S}$ is

$$A(\mathcal{S}) = \max_{Q \in \mathcal{Q}} A(Q)$$

Note that $A(\mathcal{S})$ is in the range of 1 (best) to B (worst)

- best when $|C_Q| = \left\lceil \frac{|Q|}{B} \right\rceil$, for all $Q \in \mathcal{Q}$
- worst when $|C_Q| = |Q|$, for some $Q \in \mathcal{Q}$

Models of Indexing: workload model

Example. In one-dimensional range query workloads, any access method that partitions data items along the associated linear order $\leq$ achieves optimality, both in access overhead and average redundancy.

- B+trees, with an additive constant of two or three in access overhead

Models of Indexing: workload model

The model suppresses

- how to map I into $\mathcal{S}$

Models of Indexing: workload model

The model suppresses

- how to map I into $\mathcal{S}$
- how to map $Q \in \mathcal{Q}$ into $\mathcal{S}$

Models of Indexing: workload model

The model suppresses

- how to map I into $\mathcal{S}$
- how to map $Q \in \mathcal{Q}$ into $\mathcal{S}$
- storage and access costs associated with these two (e.g., auxiliary information such as directories and internal nodes)

Models of Indexing: workload model

The model suppresses

- how to map I into $\mathcal{S}$
- how to map $Q \in \mathcal{Q}$ into $\mathcal{S}$
- storage and access costs associated with these two (e.g., auxiliary information such as directories and internal nodes)

rationale

- the model is only interested in lower bounds
- suppressed costs don't seem to be the primary source of difficulty in practice
- secondary storage techniques, such as buffer management, absorb many of these auxiliary costs

Models of Indexing: workload model

Benefits include:

- focus on main balance in indexing: the tradeoff between redundancy and access costs

Models of Indexing: workload model

Benefits include:

- focus on main balance in indexing: the tradeoff between redundancy and access costs
- can analyze new workloads by showing them to be isomorphic to well-understood workloads

Models of Indexing: GMAP model

(2) GMAP model

- introduced by Tsatalos et al, DEXA 1994, *VLDBJ* 1996
- focuses on decoupling logical schema from physical schema
  - how to map I into $\mathcal{S}$

Models of Indexing: GMAP model

- logical schema is modeled as entities, their attributes, and relationships between entities
  - as a logical collection of binary relations

Models of Indexing: GMAP model

- logical schema is modeled as entities, their attributes, and relationships between entities
  - as a logical collection of binary relations
- presents an extended conjunctive query language to define physical structures on the logical schema

Models of Indexing: GMAP model

Example. Suppose we have Faculty, Students, Departments, and Courses, and we'd like to build a b+tree on CS faculty, by their research area

Models of Indexing: GMAP model

Example. Suppose we have Faculty, Students, Departments, and Courses, and we'd like to build a b+tree on CS faculty, by their research area

```
def_gmap cs_faculty_by_area as btree by
  given Faculty.area
  select Faculty
  where Faculty worksIn Dept
        and Dept = csOID
```

Models of Indexing: GMAP model

Example. Suppose we have Faculty, Students, Departments, and Courses, and we'd like to build a b+tree on CS faculty, by their research area

$$\pi_{F,F.area}(\sigma_{D=csOID}(F.area \bowtie worksIn))$$

Models of Indexing: GMAP model

Then, given query/update Q on the logical schema, translate Q into query/update over physical structure

Models of Indexing: GMAP model

Then, given query/update Q on the logical schema, translate Q into query/update over physical structure

Examples

- relations (primary heap)
- secondary indexes (B+trees)
- join indexes
- access support relations
- ...

Models of Indexing: query language model

(3) Query language model

- presented in Fletcher et al, *Inf Sys* 2009
- focuses on finding ideal object partitioning for a given query language
  - how to map I into $\mathcal{S}$
  - how to map $Q \in \mathcal{Q}$ into $\mathcal{S}$

Models of Indexing: query language model

- a query language $\mathcal{L}$ is typically an infinite object
- for an instance I, $\mathcal{L}(I)$ is also typically infinite

Models of Indexing: query language model

- a query language $\mathcal{L}$ is typically an infinite object
- for an instance I, $\mathcal{L}(I)$ is also typically infinite

- however, the partition of I with respect to distinguishability of objects by queries in $\mathcal{L}$, denoted $I/\mathcal{L}$, is finite if I is finite
- so, if we can efficiently build $I/\mathcal{L}$, we have an ideal basis for indexing I with respect to $\mathcal{L}$

Models of Indexing: query language model

Examples

- keyword search and inverted indexes
  - language equivalence is captured by keyword equivalence

Models of Indexing: query language model

Examples

- keyword search and inverted indexes
  - language equivalence is captured by keyword equivalence
- range queries and B+trees
  - language equivalence is captured by value equivalence

Models of Indexing: query language model

Examples

- keyword search and inverted indexes
  - language equivalence is captured by keyword equivalence
- range queries and B+trees
  - language equivalence is captured by value equivalence
- XML path queries and "structural" indexes
  - language equivalence is captured by bisimulation equivalence

Models of Indexing: query language model

Examples

- keyword search and inverted indexes
  - language equivalence is captured by keyword equivalence
- range queries and B+trees
  - language equivalence is captured by value equivalence
- XML path queries and "structural" indexes
  - language equivalence is captured by bisimulation equivalence

All of these "structural" characterizations of language equivalence are efficiently computable

Models of Indexing: query language model

Questions

How to go from $\mathcal{L}$ to $I/\mathcal{L}$ to $\mathcal{L}(I)$?

- That is, how to map a query $q \in \mathcal{L}$ to $q(I)$, via $I/\mathcal{L}$?
  - rewrite q on reduced space $I/\mathcal{L}$

Models of Indexing: query language model

Questions

How to go from $\mathcal{L}$ to $I/\mathcal{L}$ to $\mathcal{L}(I)$?

- That is, how to map a query $q \in \mathcal{L}$ to $q(I)$, via $I/\mathcal{L}$?
  - rewrite q on reduced space $I/\mathcal{L}$

How can we use $I/\mathcal{L}$ to query in a superset of $\mathcal{L}$?

Models of Indexing: query language model

Questions

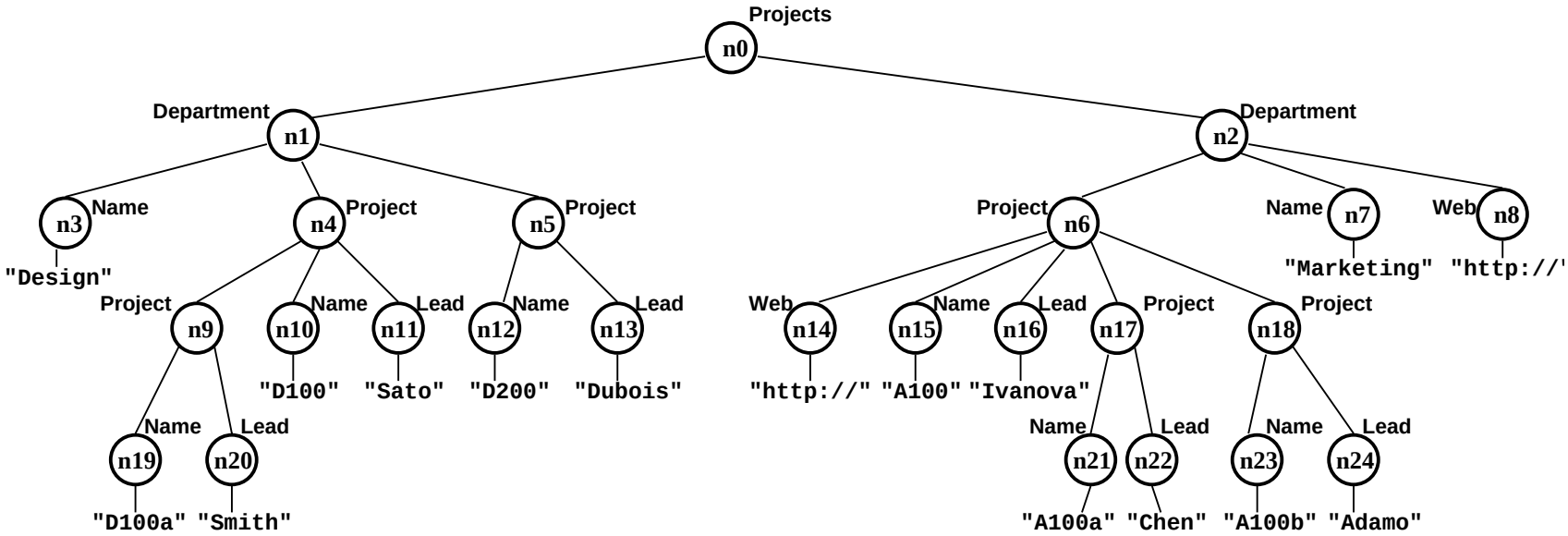
How to go from $\mathcal{L}$ to $I/\mathcal{L}$ to $\mathcal{L}(I)$?

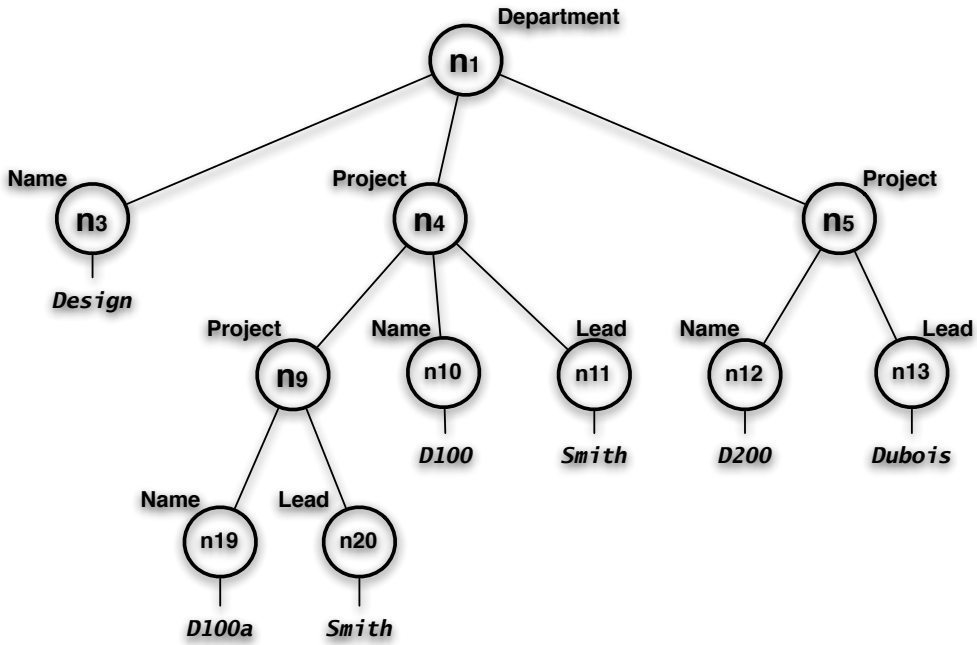
- That is, how to map a query $q \in \mathcal{L}$ to $q(I)$, via $I/\mathcal{L}$?
  - rewrite q on reduced space $I/\mathcal{L}$

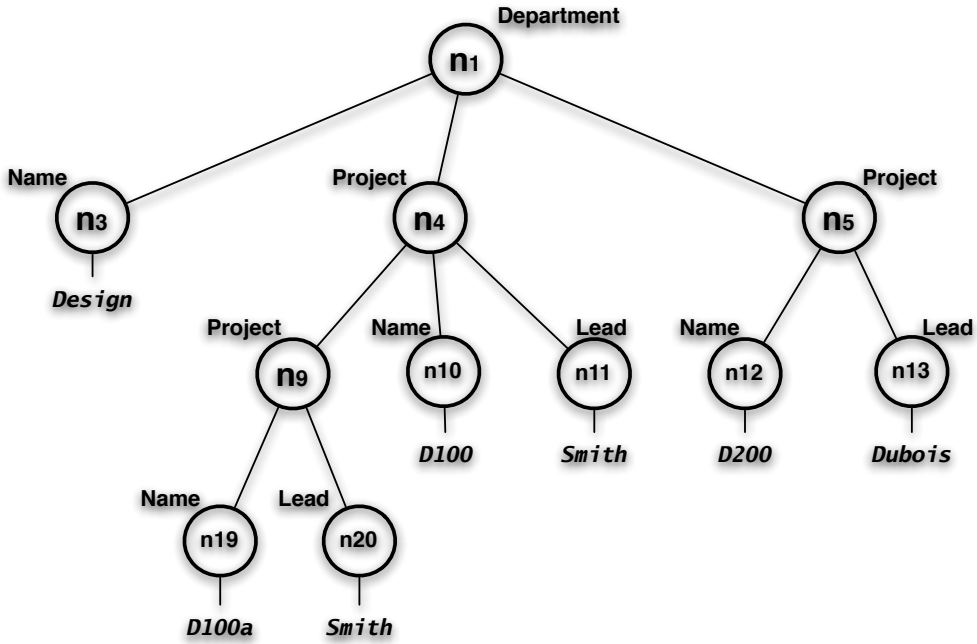
How can we use $I/\mathcal{L}$ to query in a superset of $\mathcal{L}$?

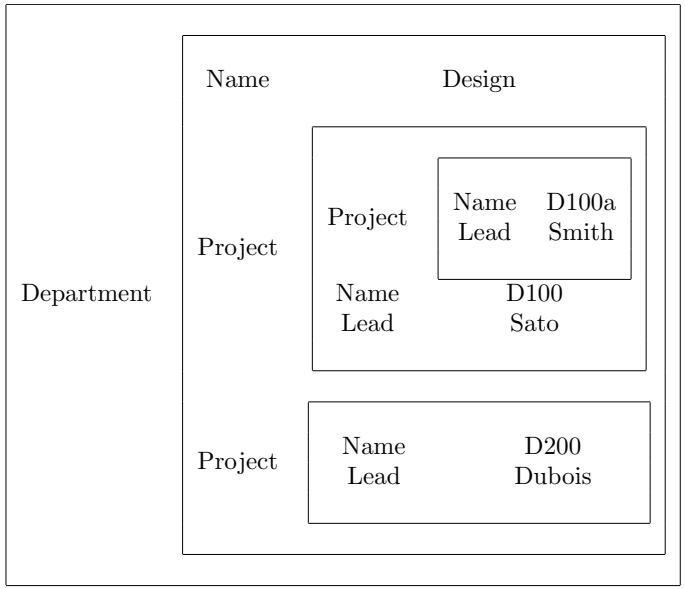
- query decomposition, mapping subexpressions to blocks, and then result reconstruction
- example: branching queries and $P(k)$ indexes for XML

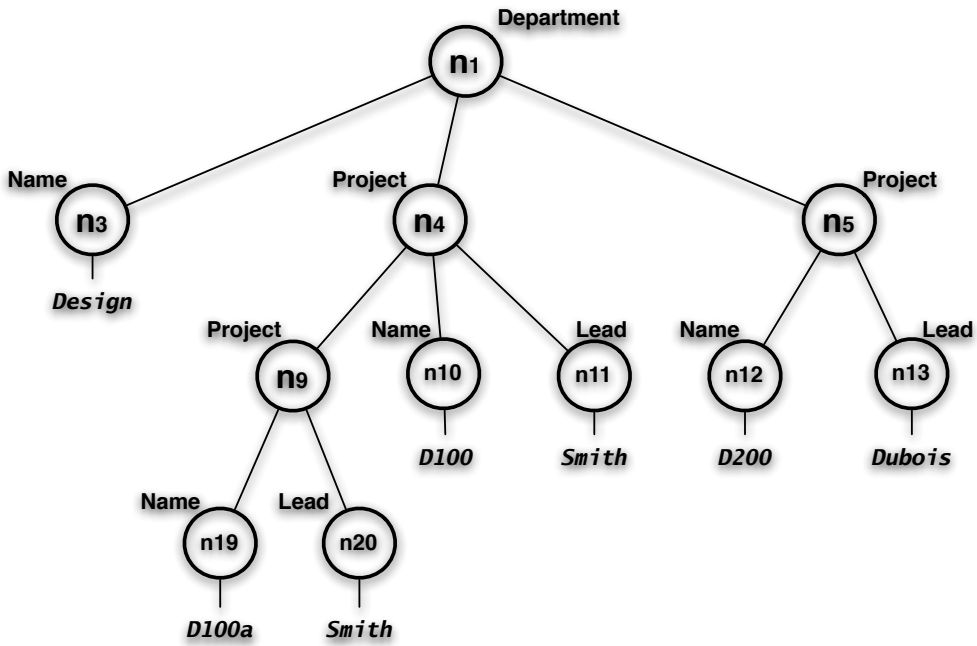
Case study: XPath and XML


Projects
n0
Department
n1
Department
n2
Name
n3
Project
n4
Project
n5
Project
n6
Name
n7
Web
n8
"Design"
"Marketing"
"http://"
Project
n9
Name
n10
Lead
n11
Name
n12
Lead
n13
Web
n14
Name
n15
Lead
n16
Project
n17
Project
n18
"D100"
"Sato"
"D200"
"Dubois"
"http://"
"A100"
"Ivanova"
Name
n19
Lead
n20
Name
n21
Lead
n22
Name
n23
Lead
n24
"D100a"
"Smith"
"A100a"
"Chen"
"A100b"
"Adamo"


Department
n1
Name
n3
Design
Project
n4
Project
n5
Project
n9
Name
n10
D100
Lead
n11
Smith
Name
n12
D200
Lead
n13
Dubois
Name
n19
D100a
Lead
n20
Smith


Department
n1
Name
n3
Design
Project
n4
Project
n5
Project
n9
Name
n10
D100
Lead
n11
Smith
Name
n12
D200
Lead
n13
Dubois
Name
n19
D100a
Lead
n20
Smith


Department
Name
Design
Project
Project
Name
Lead
D100a
Smith
Name
Lead
D100
Sato
Project
Name
Lead
D200
Dubois




```
<Department>
  <Name>Design</Name>
  <Project>
    <Project>
      <Name>D100a</Name>
      <Lead>Smith</Name>
    </Project>
    <Name>D100</Name>
    <Lead>Sato</Lead>
  </Project>
  <Project>
    <Name>D200</Name>
    <Lead>Dubois</Lead>
  </Project>
</Department>
```

XML Data Model

Documents $D = (V, Ed, r, \lambda)$ are finite unordered node-labeled trees:

- nodes V
- edges $Ed \subseteq V \times V$
- root r
- labels $\lambda : V \to L$

XPath

Expressions specify tree patterns

XPath

Expressions specify tree patterns

example: “Retrieve leaders of subprojects.”

XPath

Expressions specify tree patterns

example: “Retrieve leaders of subprojects.”

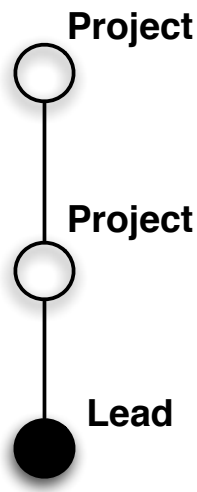
```
//Project/Project/Lead
```

XPath

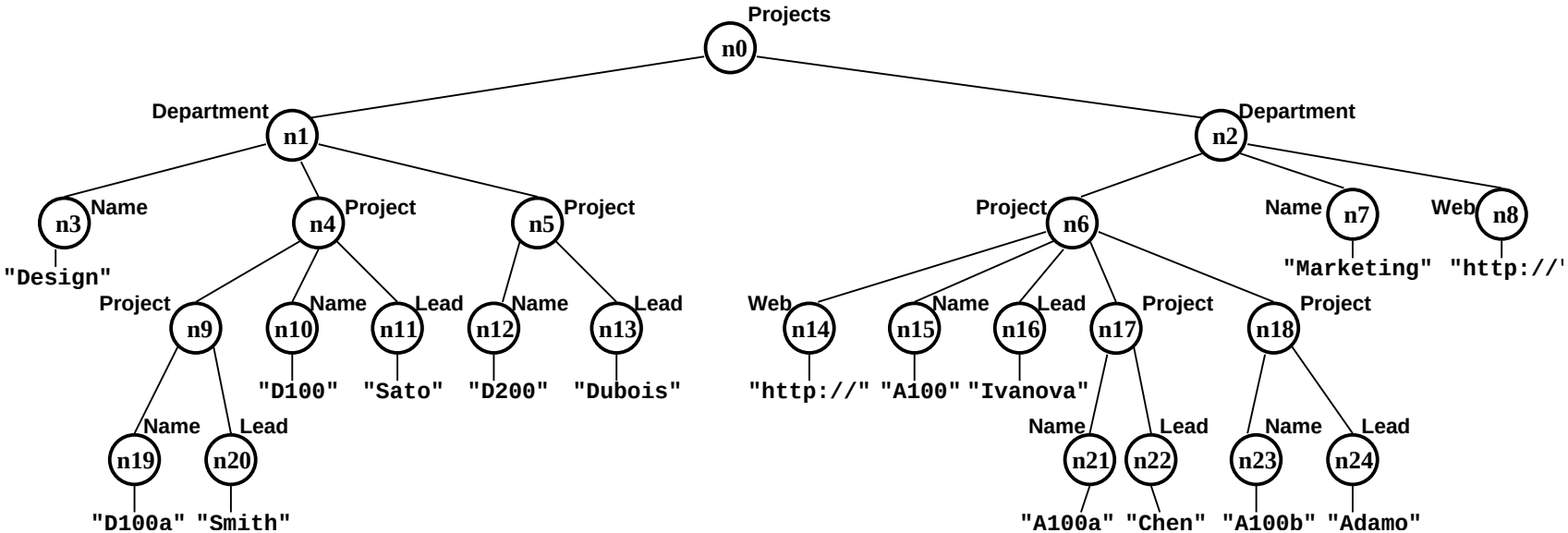
Expressions specify tree patterns

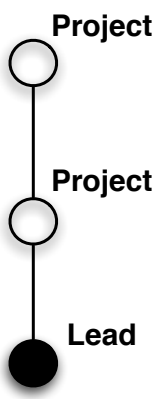
example: "Retrieve leaders of subprojects."

```
//Project/Project/Lead
```

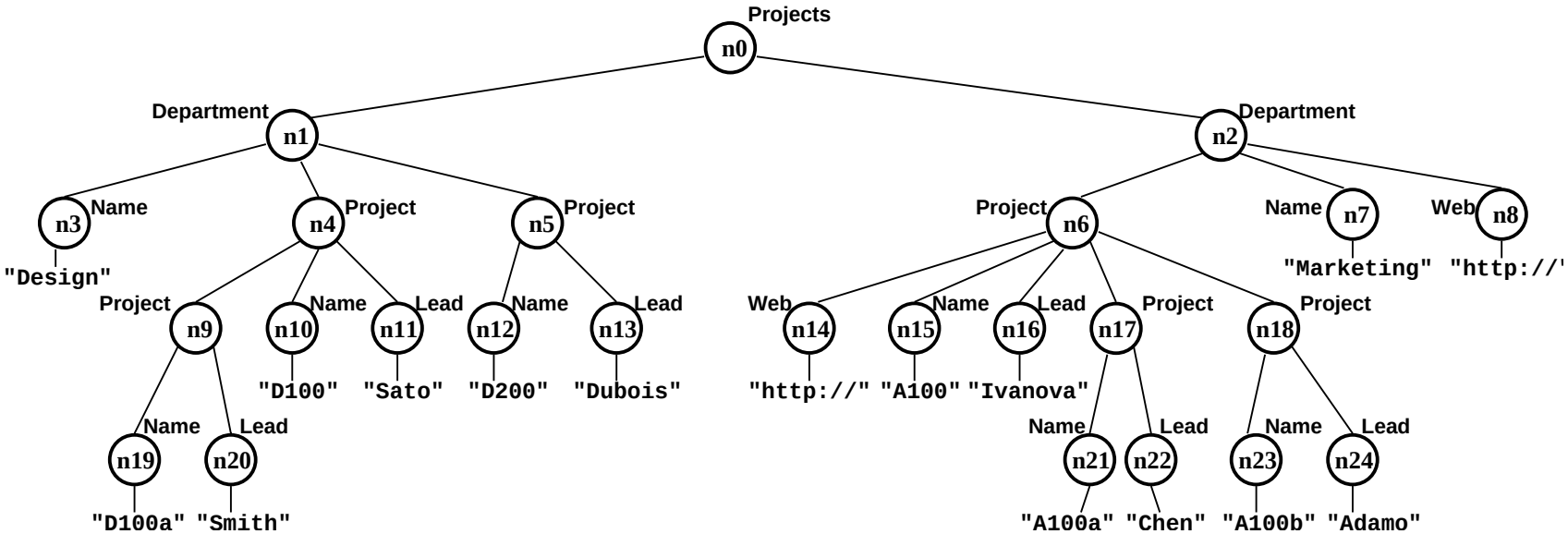


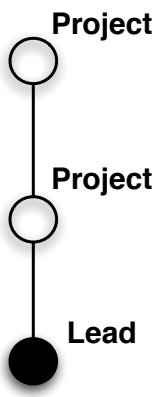
XPath


Projects
n0
Department
n1
Department
n2
Name
n3
Project
n4
Project
n5
Project
n6
Name
n7
Web
n8
"Design"
"Marketing"
"http://'
Project
n9
Name
n10
Lead
n11
Name
n12
Lead
n13
Web
n14
Name
n15
Lead
n16
Project
n17
Project
n18
"D100"
"Sato"
"D200"
"Dubois"
"http://"
"A100"
"Ivanova"
Name
n19
Lead
n20
Name
n21
Lead
n22
Name
n23
Lead
n24
"D100a"
"Smith"
"A100a"
"Chen"
"A100b"
"Adamo"


Project
Project
Lead

XPath





$$\longrightarrow \quad \{n_{20}, n_{22}, n_{24}\}$$

XPath

Expressions specify tree patterns

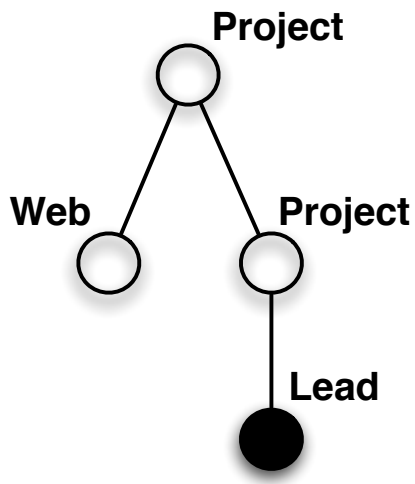
example: “Retrieve leaders of a subproject of a project having a website.”

XPath

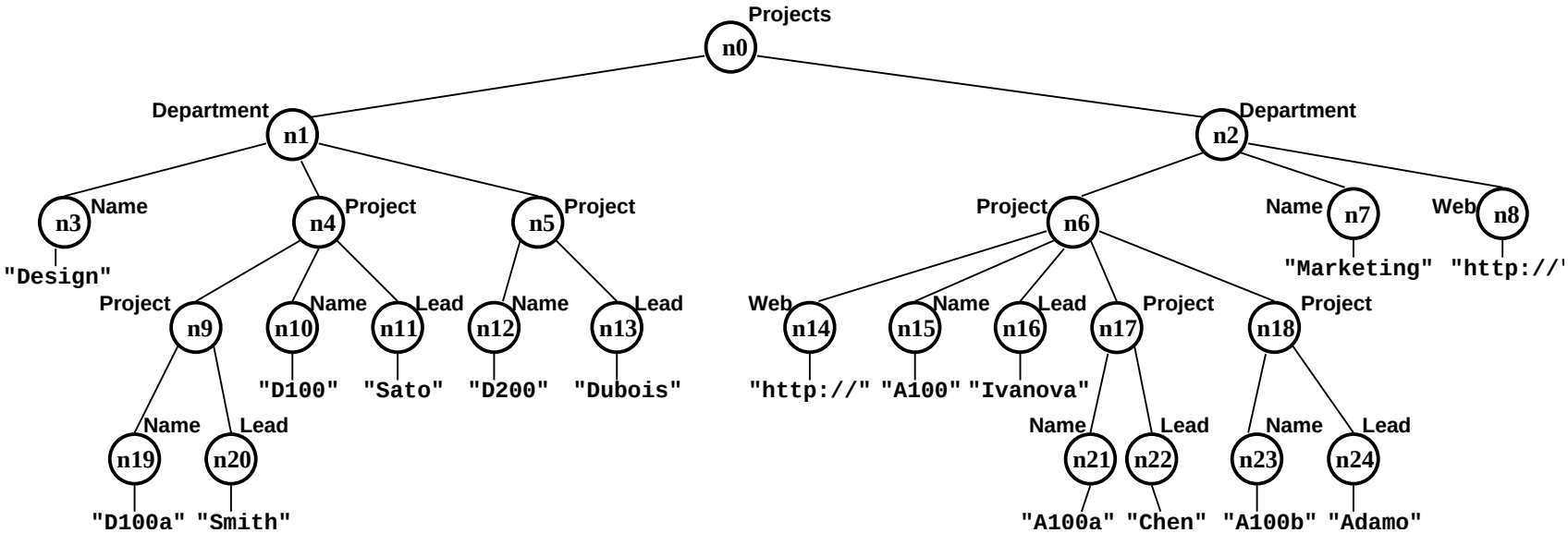
Expressions specify tree patterns

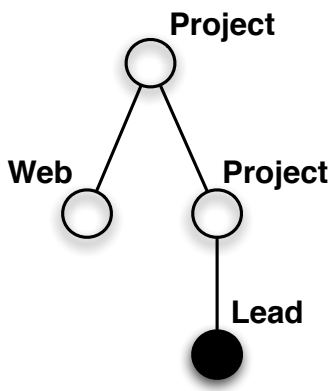
example: "Retrieve leaders of a subproject of a project having a website."

```
//Project[/Web]/Project/Lead
```

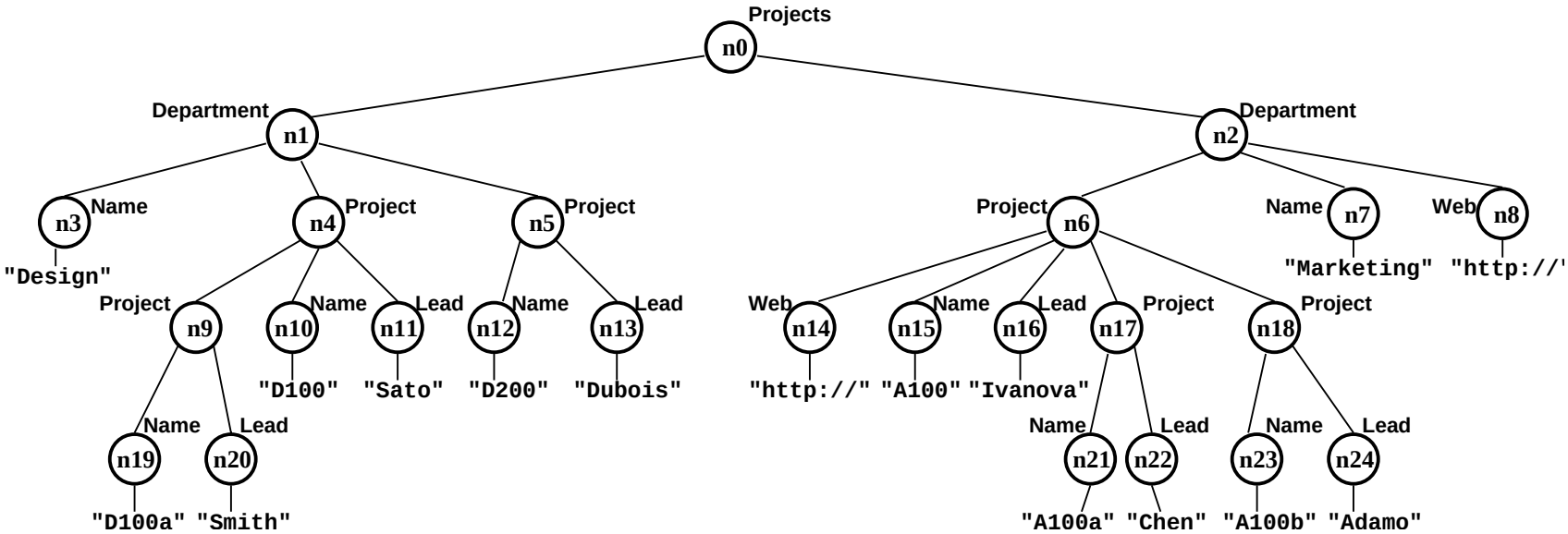


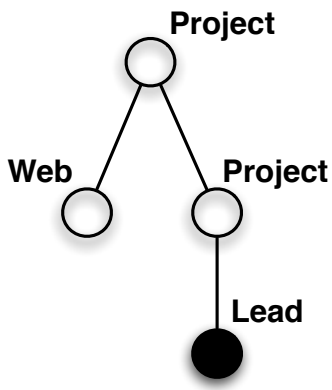
XPath


Projects
n0
Department
n1
Department
n2
Name
n3
Project
n4
Project
n5
Project
n6
Name
n7
Web
n8
"Design"
"Marketing"
"http://'
Project
n9
Name
n10
Lead
n11
Name
n12
Lead
n13
Web
n14
Name
n15
Lead
n16
Project
n17
Project
n18
"D100"
"Sato"
"D200"
"Dubois"
"http://"
"A100"
"Ivanova"
Name
n19
Lead
n20
Name
n21
Lead
n22
Name
n23
Lead
n24
"D100a"
"Smith"
"A100a"
"Chen"
"A100b"
"Adamo"


Project
Web
Project
Lead

XPath





$$\longrightarrow \quad \{n_{22}, n_{24}\}$$

The XPath algebra

Given document

$$D = (V, Ed, r, \lambda)$$

The XPath algebra

Given document

$$D = (V, Ed, r, \lambda)$$

$$\begin{aligned}
\varepsilon(D) &= \{(n, n) \mid n \in V\} \\
\emptyset(D) &= \emptyset \\
\ell(D) &= \{(n, n) \mid m \in V \text{ and } \lambda(n) = \ell\} \\
\downarrow(D) &= Ed \\
\uparrow(D) &= Ed^{-1}
\end{aligned}$$

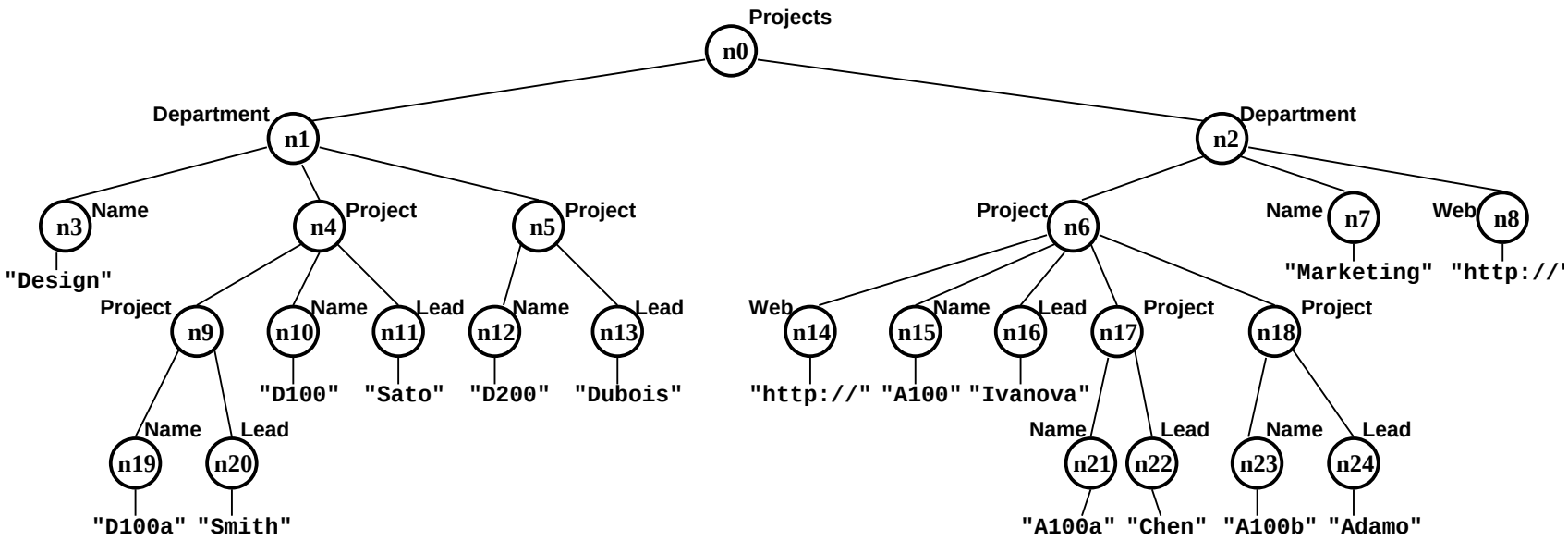
The XPath algebra

Given document

$$D = (V, Ed, r, \lambda)$$

$$\begin{aligned}
\varepsilon(D) &= \{(n,n) \mid n \in V\} \\
\emptyset(D) &= \emptyset \\
\ell(D) &= \{(n,n) \mid m \in V \text{ and } \lambda(n) = \ell\} \\
\downarrow(D) &= Ed \\
\uparrow(D) &= Ed^{-1} \\
E \cup F(D) &= E(D) \cup F(D) \\
E \cap F(D) &= E(D) \cap F(D) \\
E - F(D) &= E(D) - F(D) \\
E \circ F(D) &= \{(n,m) \mid \exists w\colon (n,w) \in E(D)\ \&\ (w,m) \in F(D)\} \\
E[F](D) &= \{(n,m) \in E(D) \mid \exists w\colon (m,w) \in F(D)\}
\end{aligned}$$

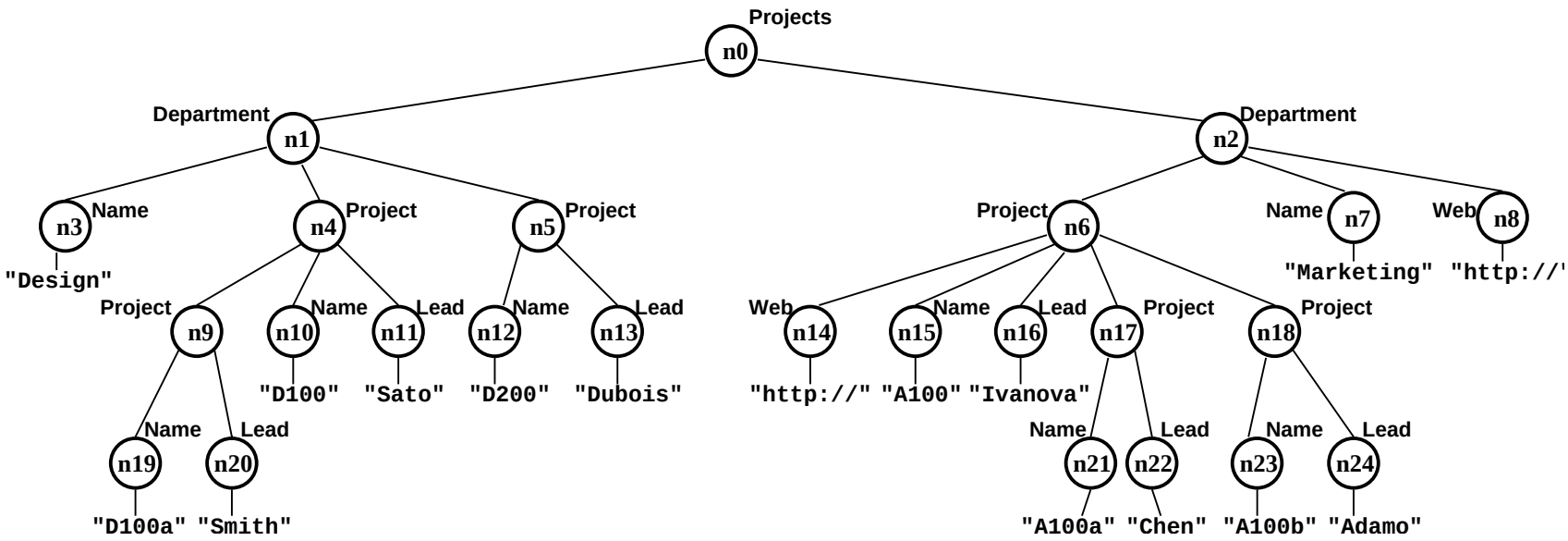
The XPath algebra



"Retrieve department names."

$$E = Projects \circ \downarrow \circ Department \circ \downarrow \circ Name$$

The XPath algebra



"Retrieve department names."

$$E = Projects \circ \downarrow \circ Department \circ \downarrow \circ Name$$

$$E(D) = \{(n_0, n_3), (n_0, n_7)\}$$

The $A(k)$ Partition of a Document

Towards structural indexing for efficient XPath evaluation ...

For nodes n and m, we have that they are $A(k)$-equivalent (denoted $n \equiv_{A(k)} m$) if

- they have the same label, and
- for $k > 0$, if one has a parent, so does the other and, furthermore, their parents are $A(k-1)$-equivalent

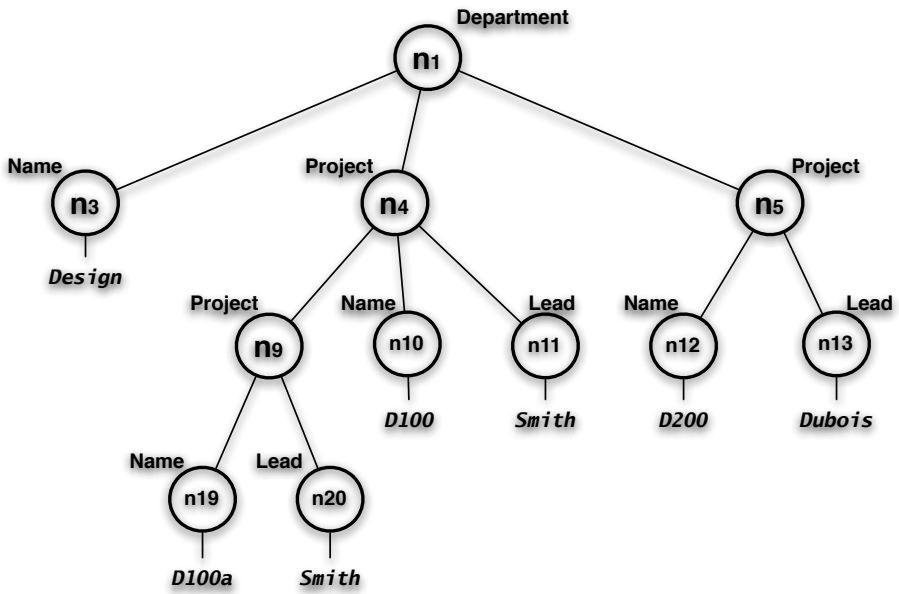
The $A(k)$ Partition of a Document

Towards structural indexing for efficient XPath evaluation ...

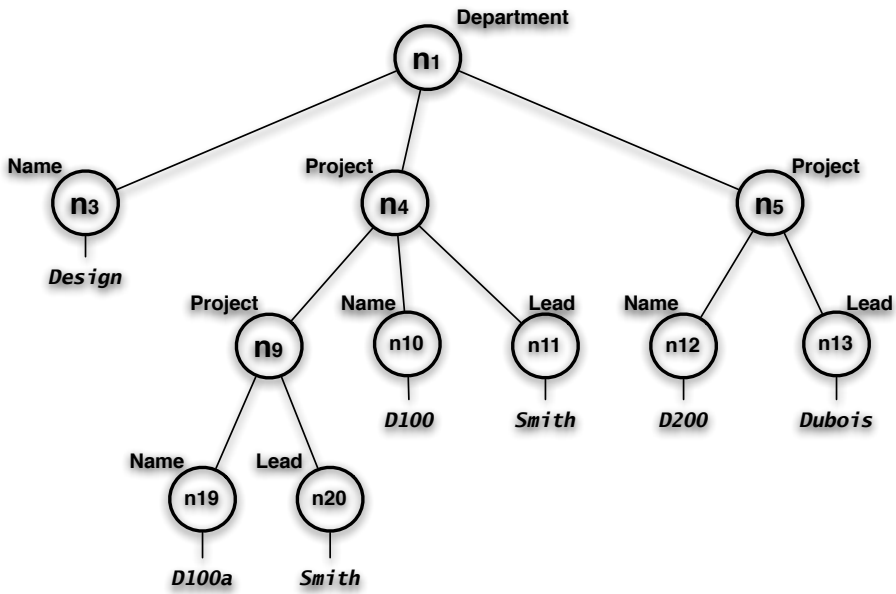
For nodes n and m, we have that they are $A(k)$-equivalent (denoted $n \equiv_{A(k)} m$) if

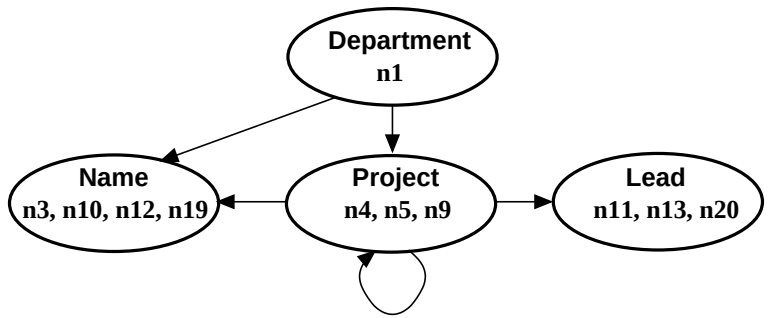
- they have the same label, and
- for $k > 0$, if one has a parent, so does the other and, furthermore, their parents are $A(k-1)$-equivalent

The partition of V induced by this (equivalence) relation on nodes is called the $A(k)$ partition of the document

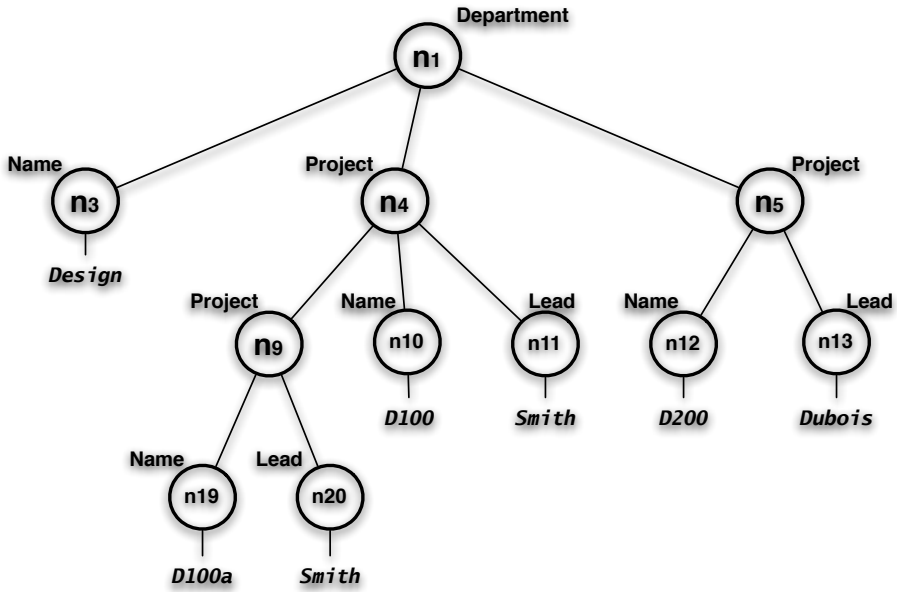


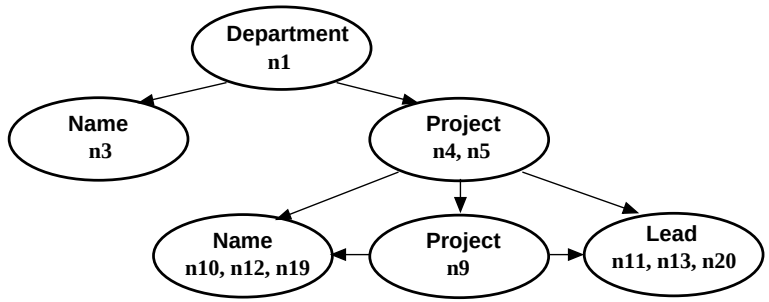
Consider $A(k)$ indexes on the "Design" department subtree


Department
n1
Name
n3
Design
Project
n4
Project
n5
Project
n9
Name
n10
D100
Lead
n11
Smith
Name
n12
D200
Lead
n13
Dubois
Name
n19
D100a
Lead
n20
Smith

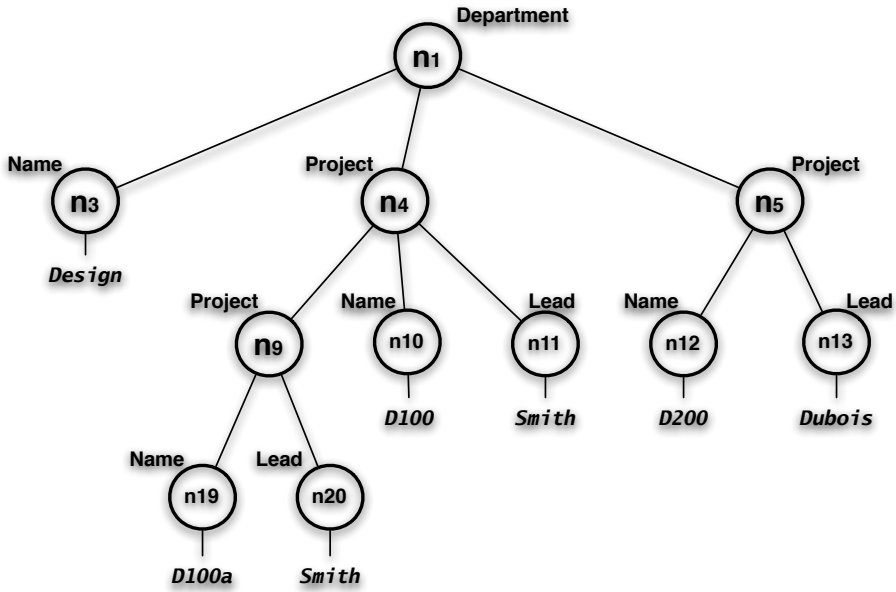

Department
n1
Name
n3, n10, n12, n19
Project
n4, n5, n9
Lead
n11, n13, n20

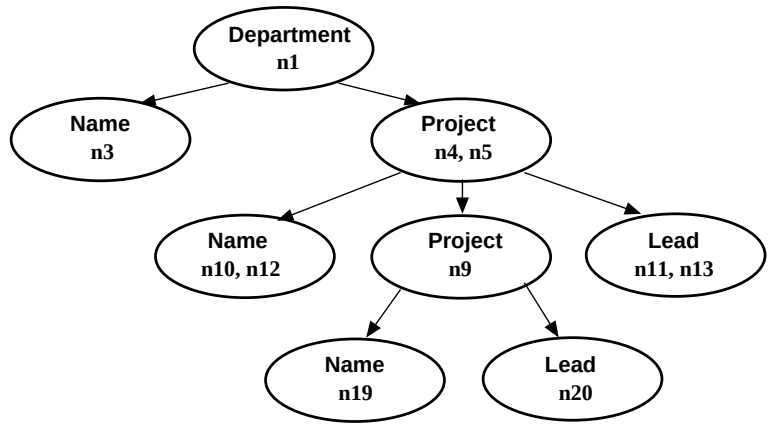
$A(0)$ index


Department
n1
Name
n3
Design
Project
n4
Project
n5
Project
n9
Name
n10
D100
Lead
n11
Smith
Name
n12
D200
Lead
n13
Dubois
Name
n19
D100a
Lead
n20
Smith


Department
n1
Name
n3
Project
n4, n5
Name
n10, n12, n19
Project
n9
Lead
n11, n13, n20

$A(1)$ index


Department
n1
Name
n3
Design
Project
n4
Project
n5
Project
n9
Name
n10
D100
Lead
n11
Smith
Name
n12
D200
Lead
n13
Dubois
Name
n19
D100a
Lead
n20
Smith


Department
n1
Name
n3
Project
n4, n5
Name
n10, n12
Project
n9
Lead
n11, n13
Name
n19
Lead
n20

$A(2)$ index

The $P(k)$ Partition of a Document

For nodes $n_1, n_2, m_1, m_2 \in V$, we have that (n_1, m_1) and (n_2, m_2) are $P(k)$-equivalent (denoted $(n_1, m_1) \equiv_{P(k)} (n_2, m_2)$) if

- (n_1, m_1) and (n_2, m_2) are in $UpPaths(D, k)$
- the distance from n_1 to m_2 in the document is the same is that from n_2 to m_2, and
- $n_1 \equiv_{A(k)} n_2$

The $P(k)$ Partition of a Document

For nodes $n_1, n_2, m_1, m_2 \in V$, we have that (n_1, m_1) and (n_2, m_2) are $P(k)$-equivalent (denoted $(n_1, m_1) \equiv_{P(k)} (n_2, m_2)$) if

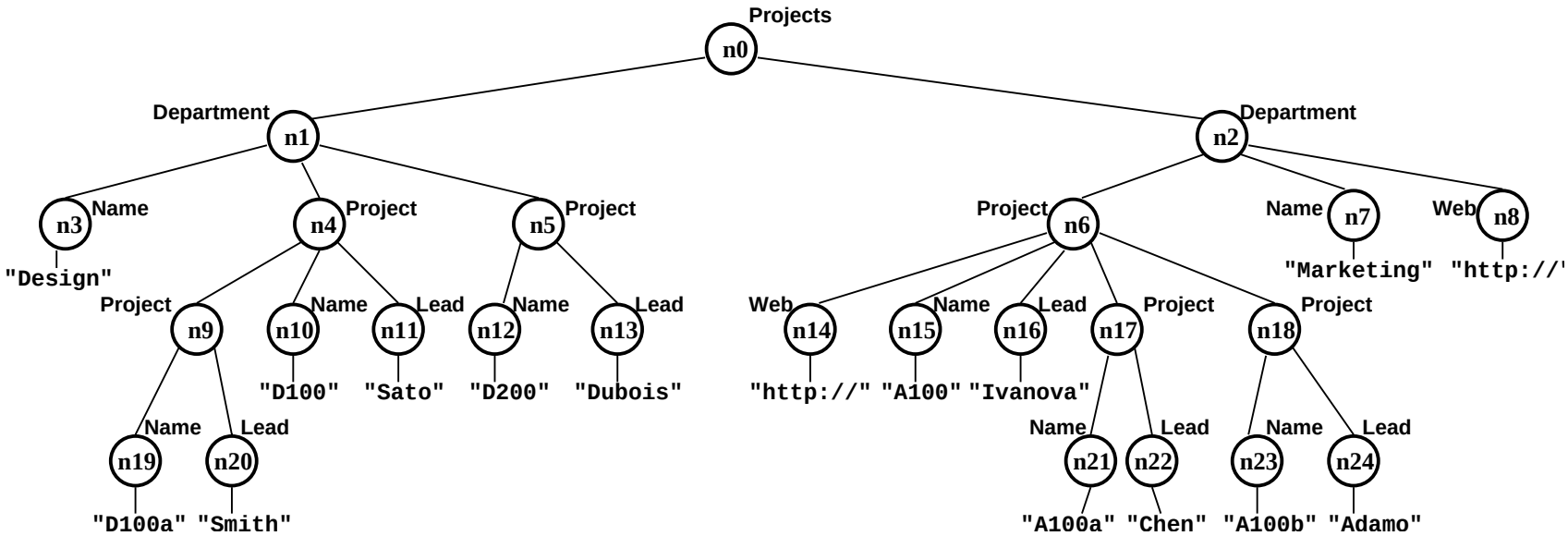
- (n_1, m_1) and (n_2, m_2) are in $UpPaths(D, k)$
- the distance from n_1 to m_2 in the document is the same is that from n_2 to m_2, and
- $n_1 \equiv_{A(k)} n_2$

The partition induced by this (equivalence) relation on node pairs in $UpPaths(D, k)$ is called the $P(k)$ partition of the document

Upward-k Algebras

For $k \geq 0$, $U(k)$ is the fragment of the XPath-Algebra with expressions that do not use the $\downarrow$ primitive and have at most k uses of the $\uparrow$ primitive in a "path"

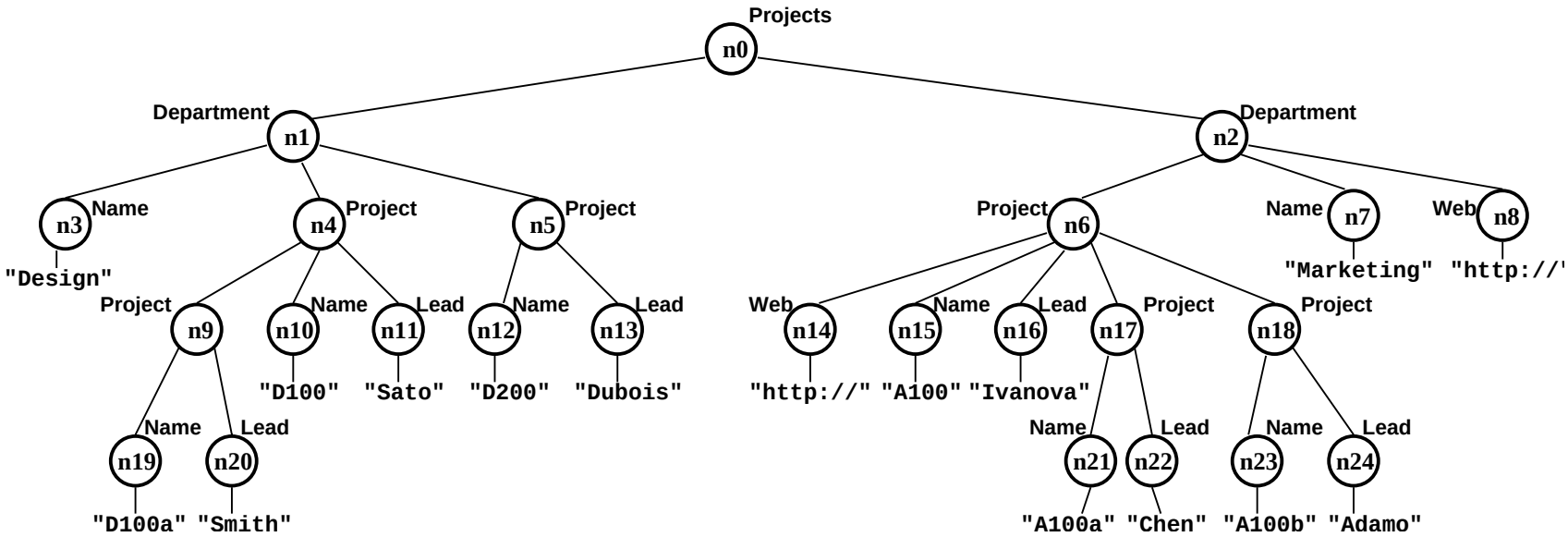
Upward-k Algebras



"Retrieve sub-project leaders."

$$E = Lead[\uparrow \circ Project \circ \uparrow \circ Project]$$

Upward-k Algebras

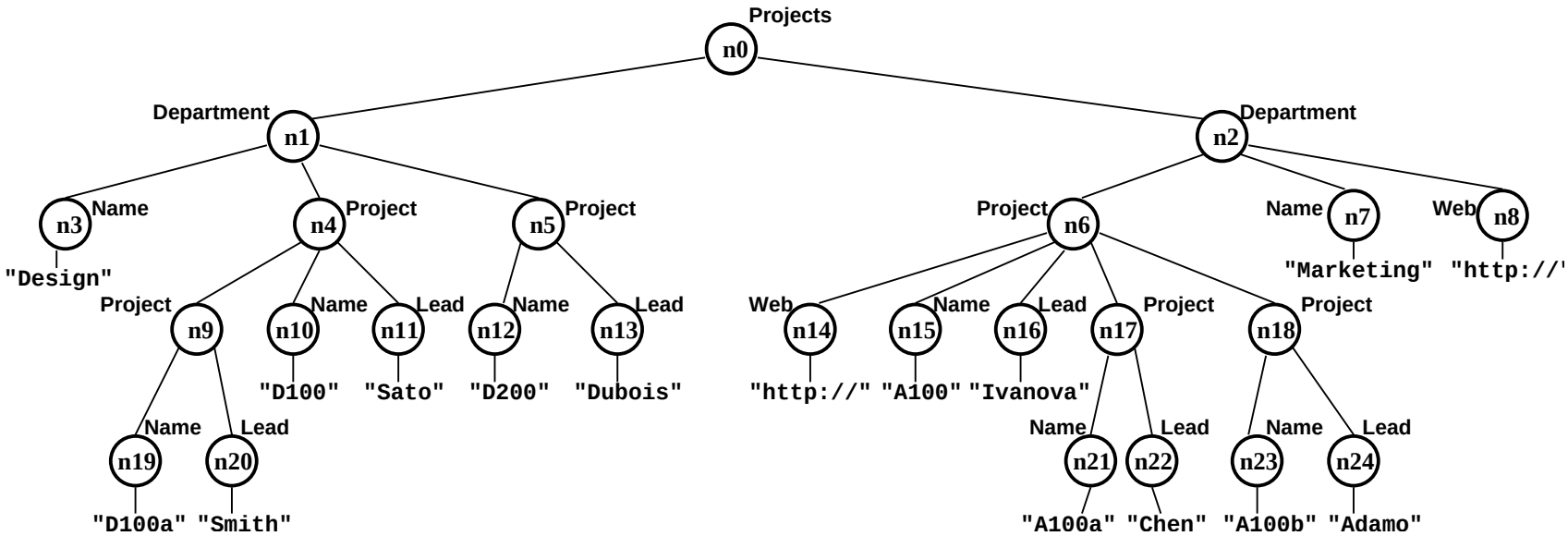


"Retrieve sub-project leaders."

$$E = Lead[\uparrow \circ Project \circ \uparrow \circ Project]$$

In $U(2)$ but not $U(1)$

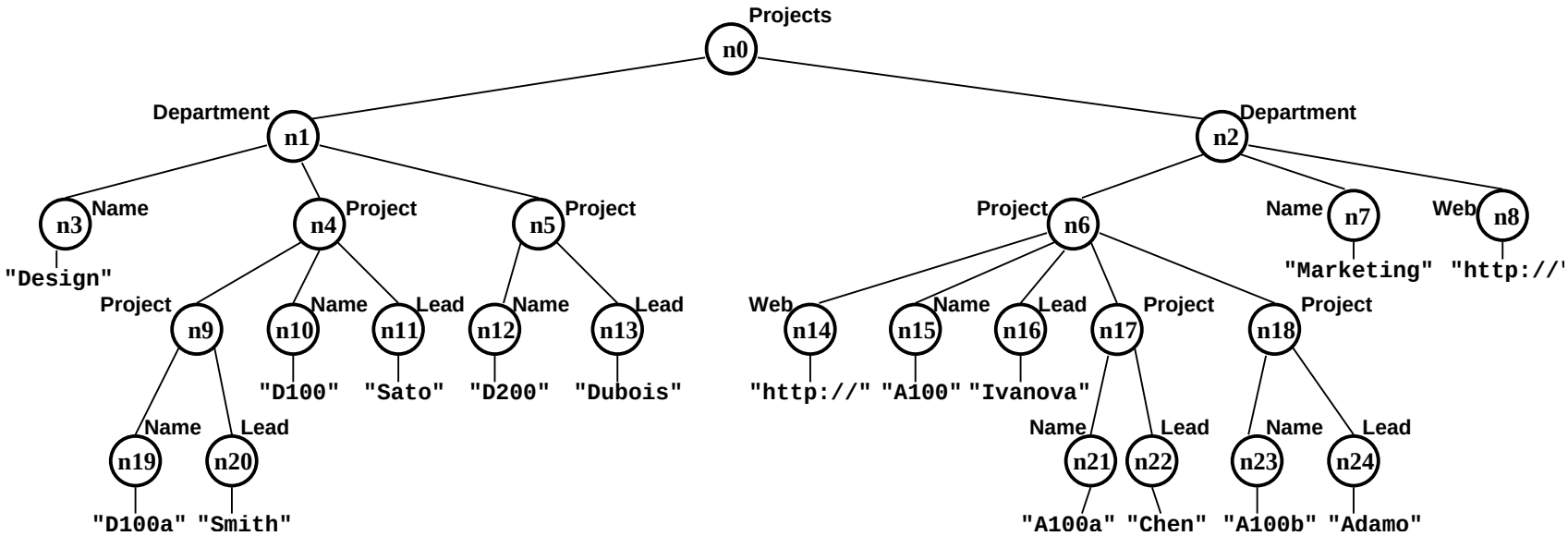
Upward-k Algebras



"Retrieve all projects which are sub-projects of projects with a website."

$$E = Project[\uparrow \circ Project \circ \downarrow \circ Web]$$

Upward-k Algebras



"Retrieve all projects which are sub-projects of projects with a website."

$$E = Project[\uparrow \circ Project \circ \downarrow \circ Web]$$

Not in $U(k)$, for any k

Language Indistinguishability

For fragment $\mathcal{L}$ of the XPath algebra, we say node pairs (n_1, m_1) and (n_2, m_2) are indistinguishable by $\mathcal{L}$ if for any expression in $e \in \mathcal{L}$, it is the case that $(n_1, m_1) \in e(D)$ if and only if $(n_2, m_2) \in e(D)$.

Language Indistinguishability

For fragment $\mathcal{L}$ of the XPath algebra, we say node pairs (n_1, m_1) and (n_2, m_2) are indistinguishable by $\mathcal{L}$ if for any expression in $e \in \mathcal{L}$, it is the case that $(n_1, m_1) \in e(D)$ if and only if $(n_2, m_2) \in e(D)$.

The partition induced by $\equiv_{\mathcal{L}}$ on $Paths(D)$ is called the $\mathcal{L}$-partition of D.

Coupling $P(k)$ and $U(k)$

Theorem (Coupling)

Let D be a document and $k \in \mathbb{N}$. The $P(k)$-partition of D and the $U(k)$-partition of D are the same.

Coupling $P(k)$ and $U(k)$

Theorem (Coupling)

Let D be a document and $k \in \mathbb{N}$. The $P(k)$-partition of D and the $U(k)$-partition of D are the same.

Theorem (Block-Union)

Let D be a document, $k \in \mathbb{N}$, and $e \in U(k)$. Then there exists a class $\mathfrak{B}_e$ of blocks of the $P(k)$-partition of D such that $e(D) = \bigcup_{B \in \mathfrak{B}_e} B$.

Coupling $P(k)$ and $U(k)$

The Coupling Theorem provides a precise linguistic characterization of the $P(k)$ partition (alternatively, a precise structural characterization of $U(k)$).

Coupling $P(k)$ and $U(k)$

The Coupling Theorem provides a precise linguistic characterization of the $P(k)$ partition (alternatively, a precise structural characterization of $U(k)$).

The Block-Union Theorem provides us with the evaluation strategy for expressions in $U(k)$. In particular, expression evaluation in $U(k)$ amounts to unions of direct index lookups.

Expression evaluation in Upward Algebra

For expressions in $U(l)$, for $l > k$, then evaluation is via

- decomposition into $U(k)$ sub-expressions
- and then joining intermediate results.

Expression evaluation in XPath Algebra

In richer fragments of the algebra, we take a two step rewriting process:

1. predicate elimination
2. invert remaining "downward" subexpressions into $U(k)$ expressions

Expression evaluation in XPath Algebra

In richer fragments of the algebra, we take a two step rewriting process:

1. predicate elimination
2. invert remaining "downward" subexpressions into $U(k)$ expressions

... and then proceed as before with $U(k)$ evaluation, to make optimal use of available indices.

Expression evaluation in XPath Algebra

Example. Suppose we have document D, the $P(2)$ partition of D, and the query $\downarrow[\downarrow]$.

Expression evaluation in XPath Algebra

Example. Suppose we have document D, the $P(2)$ partition of D, and the query $\downarrow[\downarrow]$.

Then ...

$$\downarrow[\downarrow](D)$$

Expression evaluation in XPath Algebra

Example. Suppose we have document D, the $P(2)$ partition of D, and the query $\downarrow[\downarrow]$.

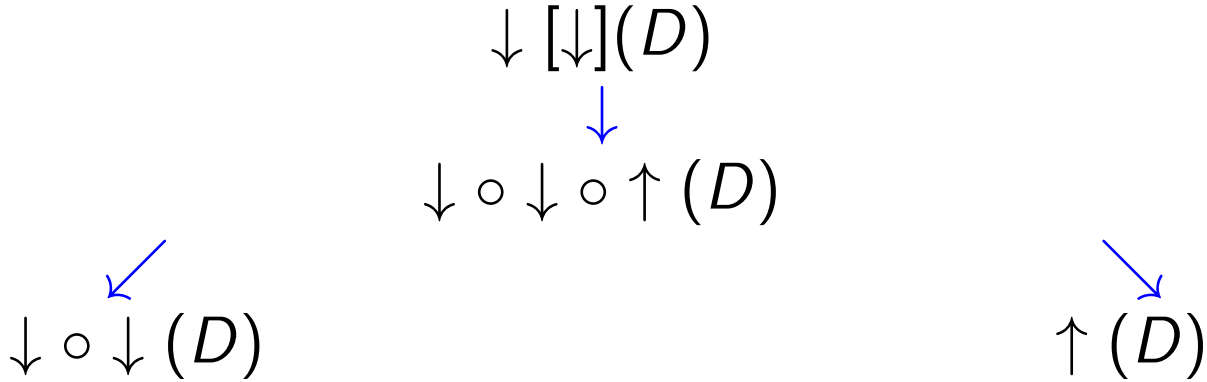
Then ...

$$\downarrow[\downarrow](D)$$
$$\downarrow$$
$$\downarrow \circ \downarrow \circ \uparrow (D)$$

Expression evaluation in XPath Algebra

Example. Suppose we have document D, the $P(2)$ partition of D, and the query $\downarrow[\downarrow]$.

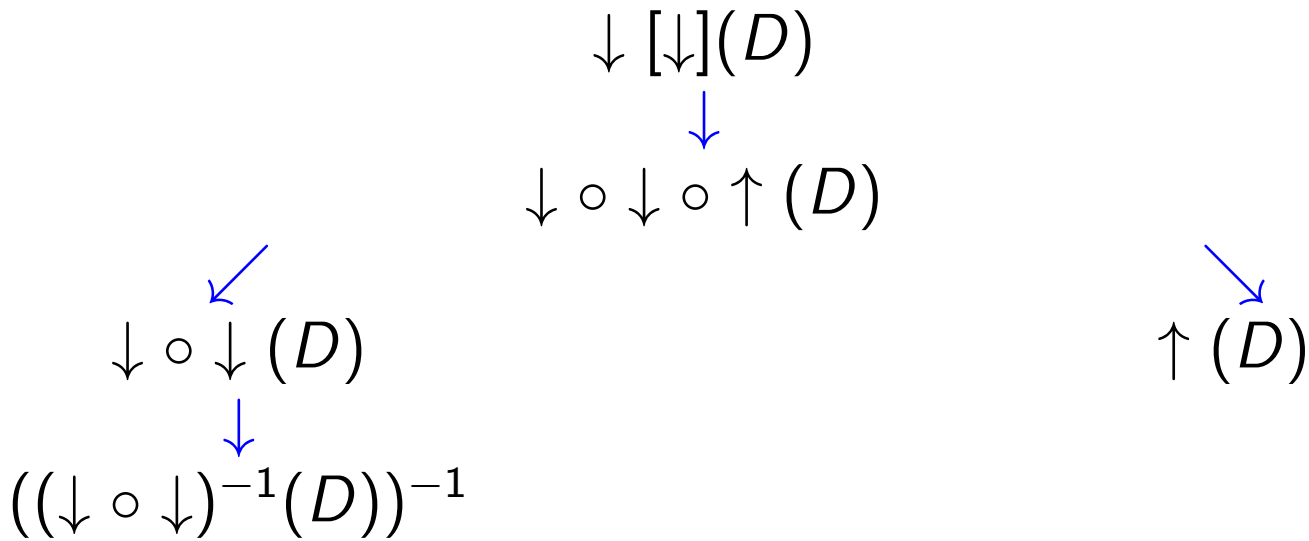
Then ...



Expression evaluation in XPath Algebra

Example. Suppose we have document D, the $P(2)$ partition of D, and the query $\downarrow[\downarrow]$.

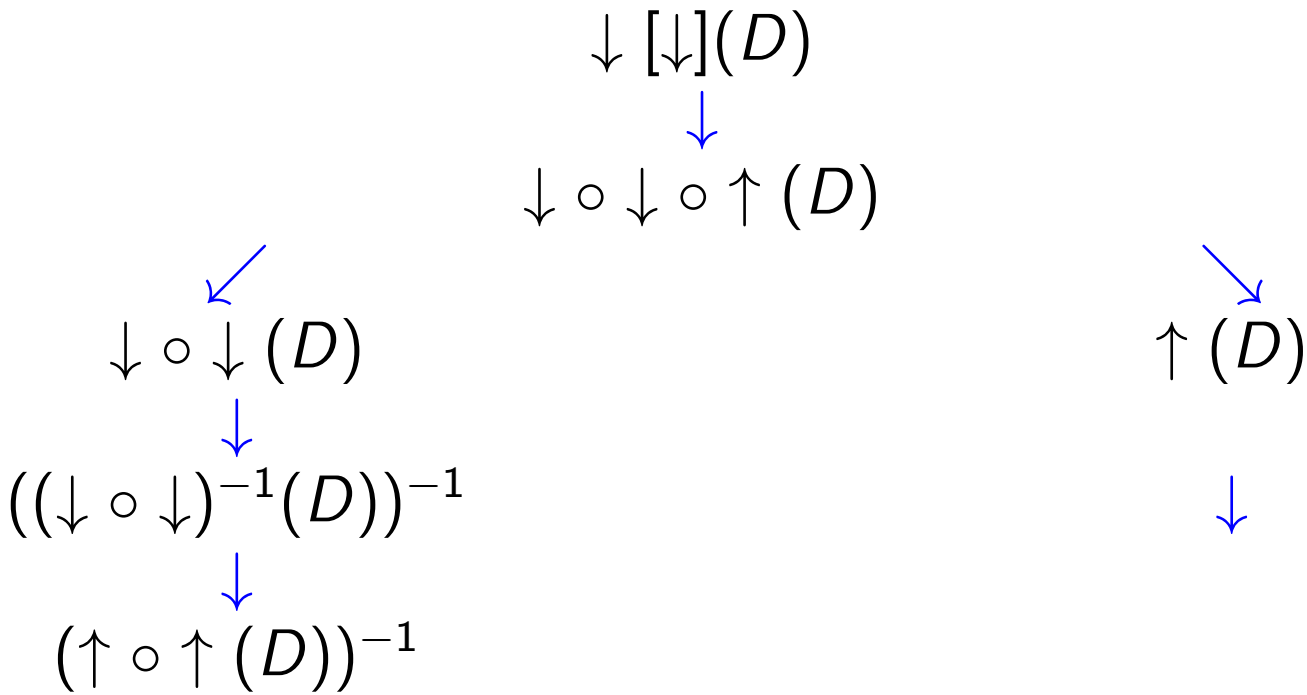
Then ...



Expression evaluation in XPath Algebra

Example. Suppose we have document D, the $P(2)$ partition of D, and the query $\downarrow[\downarrow]$.

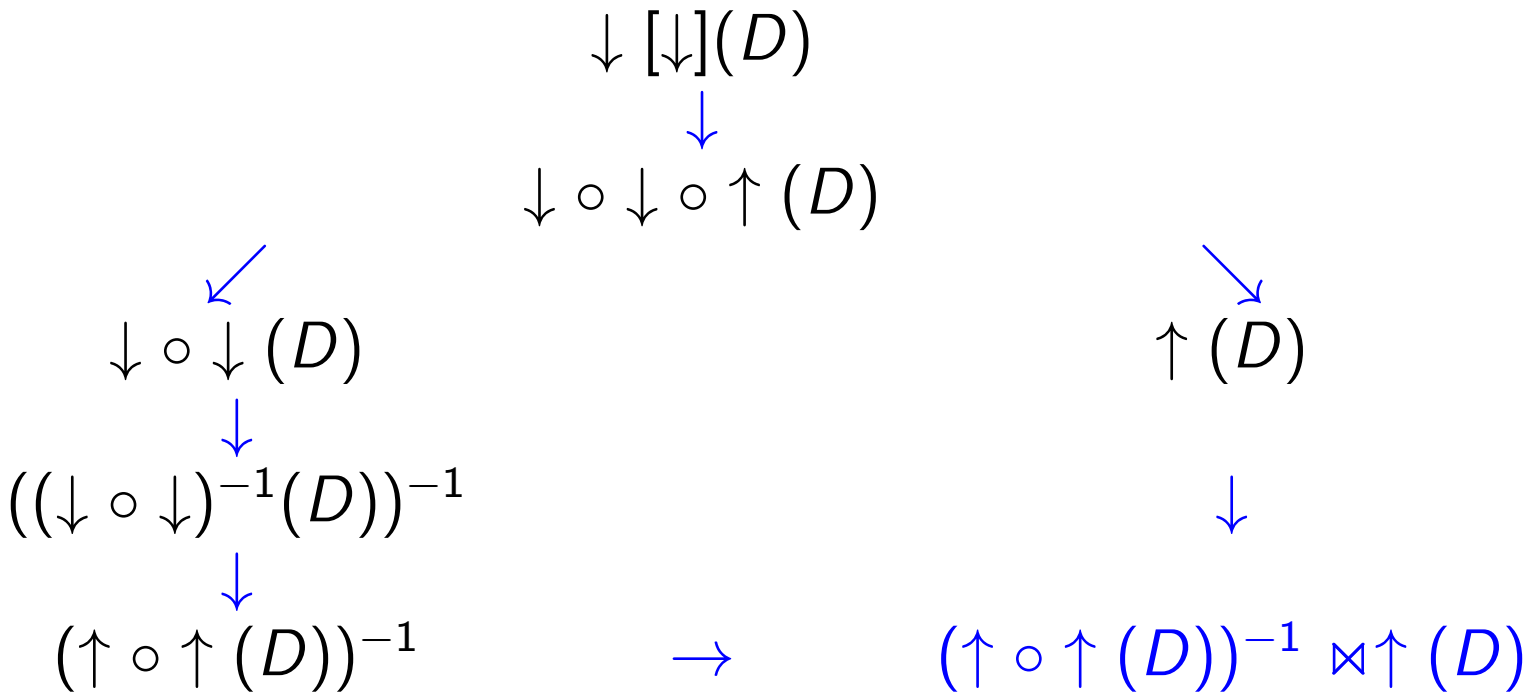
Then ...



Expression evaluation in XPath Algebra

Example. Suppose we have document D, the $P(2)$ partition of D, and the query $\downarrow[\downarrow]$.

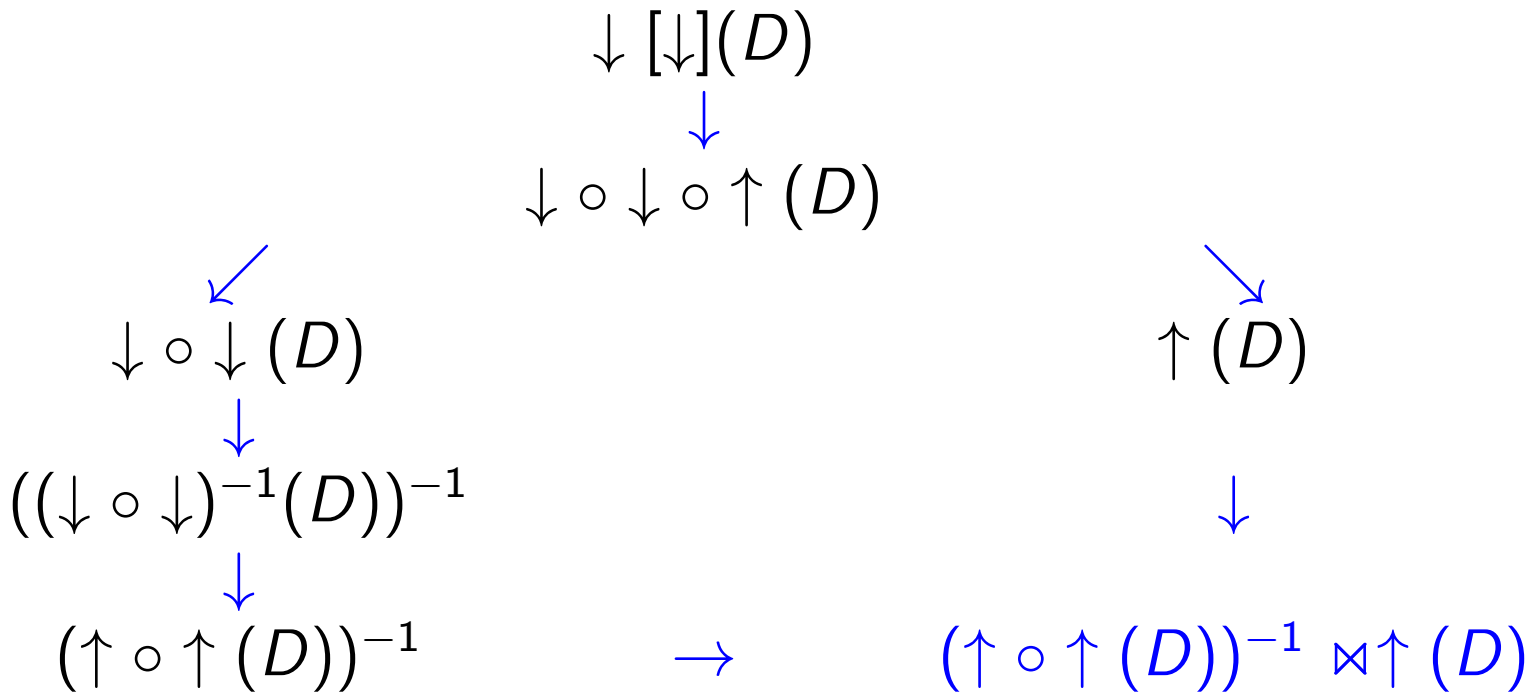
Then ...



Expression evaluation in XPath Algebra

Example. Suppose we have document D, the $P(2)$ partition of D, and the query $\downarrow[\downarrow]$.

Then ...



... and this can be evaluated directly over the $P(2)$ partition.

end of case study

Models of Indexing: query language model

Focuses on the issues set aside by the workload and GMAP models:

> *What is the relationship of a particular query language (and its syntax) to indexing?*

Models of Indexing: query language model

Focuses on the issues set aside by the workload and GMAP models:

> *What is the relationship of a particular query language (and its syntax) to indexing?*

Novel and powerful perspective, still relatively unexplored

Wrap up

Wrap up

- Indexing, part 2:
  - hash-based indexing
  - bitmap indexes
  - join indexing
  - models of indexability

Wrap up

- Indexing, part 2:
  - hash-based indexing
  - bitmap indexes
  - join indexing
  - models of indexability
- Teams and project part 2 have been posted. Find your teammates and start studying your paper.

Wrap up

- Indexing, part 2:
  - hash-based indexing
  - bitmap indexes
  - join indexing
  - models of indexability
- Teams and project part 2 have been posted. Find your teammates and start studying your paper.
- Next time (Wednesday 6 May): Query processing (i.e., evaluation of relational operators)

Credits

- Our textbook (Silberschatz *et al.*, 2011)
- Kifer *et al.*, 2006
- Valduriez, 1987