Advanced indexing techniques

Knowledge Objectives

1. Explain which index suits best depending on the selectivity of the selection predicate, the kind of comparison and the volatility of the table
2. Name three situation where an index is useless
3. Explain what a bitmap index is
4. Explain the conditions where a bitmap suits better than a B+ index and vice-versa
5. Explain what a join-index is
6. Explain the benefit of bitmap-join-indexes in multidimensional queries
7. Explain what makes the difference between a join-index and a clustered structure, from the query time point of view

Understanding Objectives

1. Know the factors involved in the choice between rebuilding an index or making individual insertions in the case of massive insertions
2. Calculate the approximate size of a bitmap index
3. Estimate the cost of a selection with a complex predicate using a bitmap index
4. Estimate the cost of a join (or semi-join) operation using a join index (either bitmap, B+, hash or cluster)
5. Given a simple query (not mixing selection and join operations) and the structures of the tables, decide whether it can be solved by accessing only the indexes
6. Given the attributes in a multi-attribute index and a complex selection predicate, decide whether the index can be used to solve the query or not

Insertion of values with an index

a) Individual insertions
- Create the index over the table and insert the tuples one by one

b) Massive insertions
- Fill the table and create the index
- Remove the index, insert tuples and rebuild the index

Algorithm to build a B-tree

1. Create a file with the entries [value,RID]
2. Sort the file by value
3. Build the leaves (filling them to the desired load)
4. Build the internal nodes (filling them to the desired load)
5. Save it to disk

Example of building a B-tree

1. Create a file with entries [value,RID]
2. Sort the file by value
3. Build the leaves (filling them to the desired load)
4. Build the internal nodes (filling them to the desired load)
5. Save it to disk

Order = 2
Load = 75%

Example of building a B-tree

1. **Create a file with entries [value,RID]**
2. Sort the file by value
3. Build the leaves (filling them to the desired load)
4. Build the internal nodes (filling them to the desired load)
5. Save it to disk

Order = 2
Load = 75%

[59,@],[3,@],[45,@],[7,@],[29,@],[8,@],[12,@],[57,@],[17,@],[1,@],[19,@],[5,@],[35,@],[53,@],[62,@],[42,@],[69,@],[25,@]

Example of building a B-tree

1. **Create a file with entries [value,RID]**
2. **Sort the file by value**
3. Build the leaves (filling them to the desired load)
4. Build the internal nodes (filling them to the desired load)
5. Save it to disk

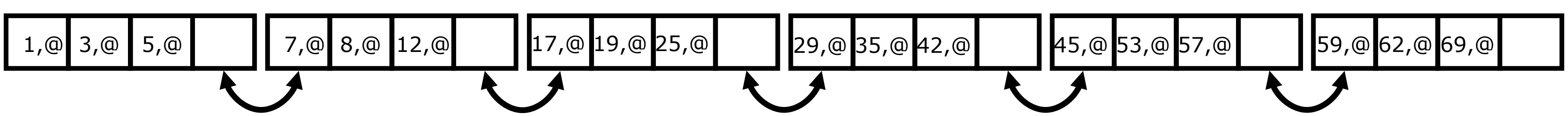
Order = 2
Load = 75%

[1,@],[3,@],[5,@],[7,@],[8,@],[12,@],[17,@],[19,@],[25,@],[29,@],[35,@],[42,@],[45,@],[53,@],[57,@],[59,@],[62,@],[69,@]

Example of building a B-tree

1. **Create a file with entries [value,RID]**
2. **Sort the file by value**
3. **Build the leaves (filling them to the desired load)**
4. Build the internal nodes (filling them to the desired load)
5. Save it to disk

Order = 2
Load = 75%

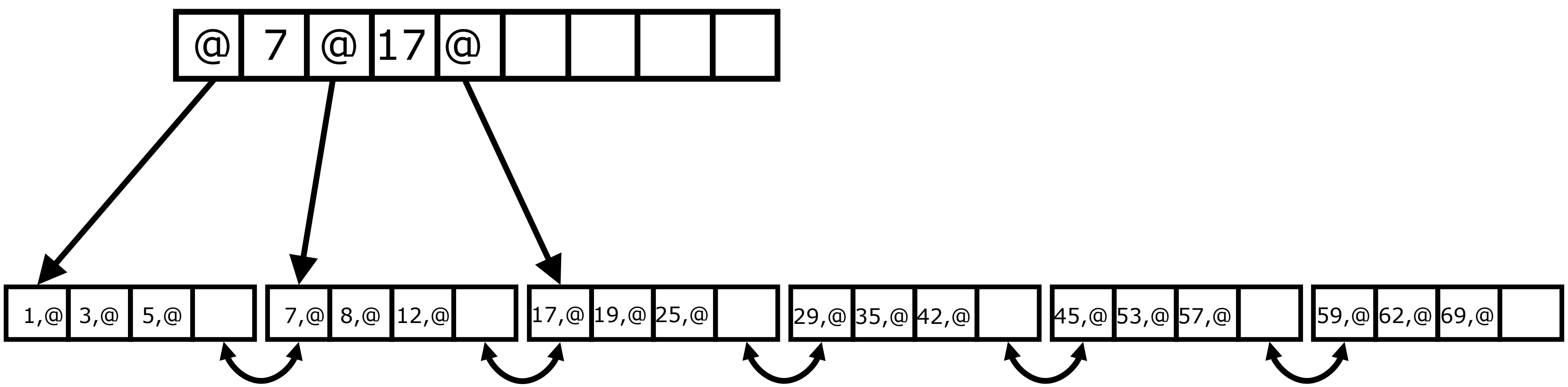


[1,@],[3,@],[5,@],[7,@],[8,@],[12,@],[17,@],[19,@],[25,@],[29,@],[35,@],[42,@],[45,@],[53,@],[57,@],[59,@],[62,@],[69,@]

Example of building a B-tree

1. **Create a file with entries [value,RID]**
2. **Sort the file by value**
3. **Build the leaves (filling them to the desired load)**
4. **Build the internal nodes (filling them to the desired load)**
5. Save it to disk

Order = 2
Load = 75%

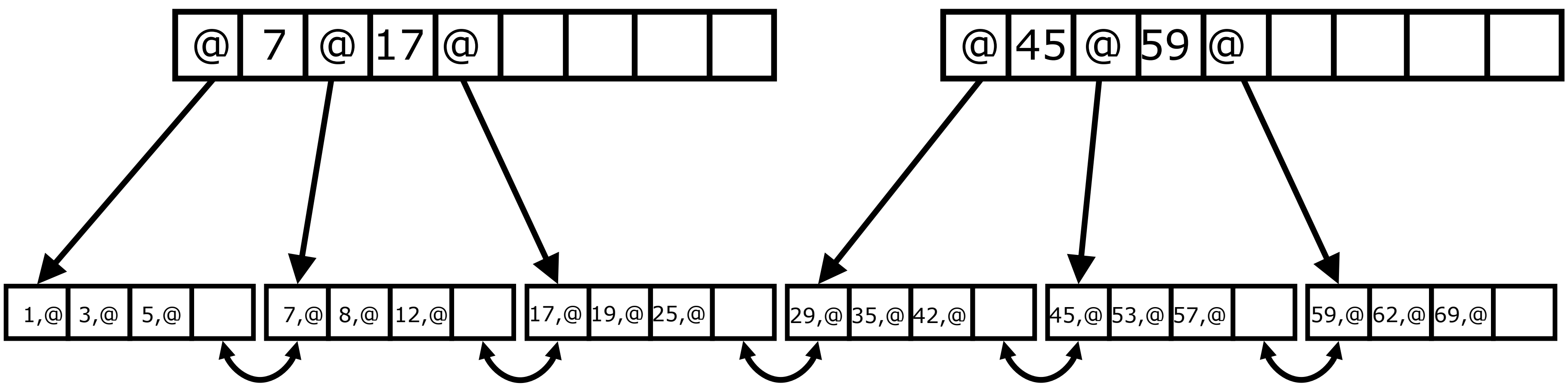


[1,@],[3,@],[5,@],[7,@],[8,@],[12,@],[17,@],[19,@],[25,@],[29,@],[35,@],[42,@],[45,@],[53,@],[57,@],[59,@],[62,@],[69,@]

Example of building a B-tree

1. **Create a file with entries [value,RID]**
2. **Sort the file by value**
3. **Build the leaves (filling them to the desired load)**
4. **Build the internal nodes (filling them to the desired load)**
5. Save it to disk

Order = 2
Load = 75%

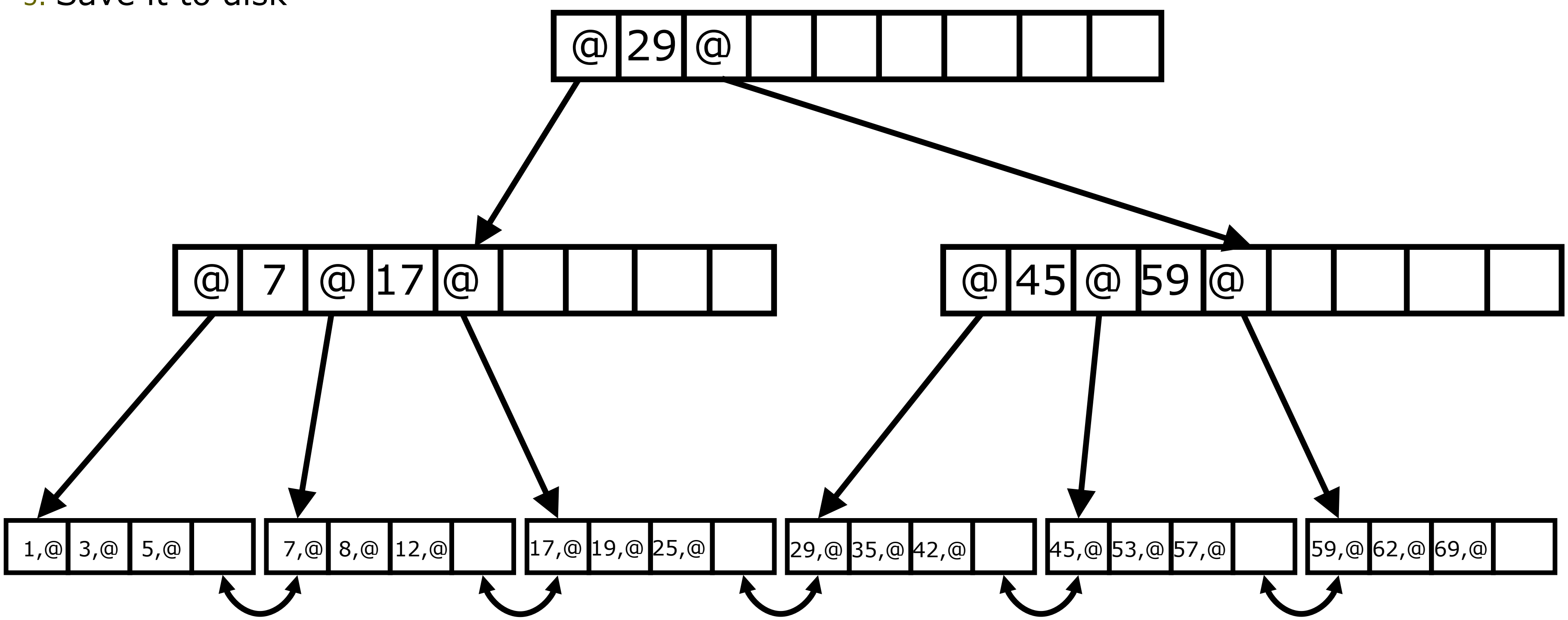


[1,@],[3,@],[5,@],[7,@],[8,@],[12,@],[17,@],[19,@],[25,@],[29,@],[35,@],[42,@],[45,@],[53,@],[57,@],[59,@],[62,@],[69,@]

Example of building a B-tree

1. **Create a file with entries [value,RID]**
2. **Sort the file by value**
3. **Build the leaves (filling them to the desired load)**
4. **Build the internal nodes (filling them to the desired load)**
5. Save it to disk

Order = 2
Load = 75%

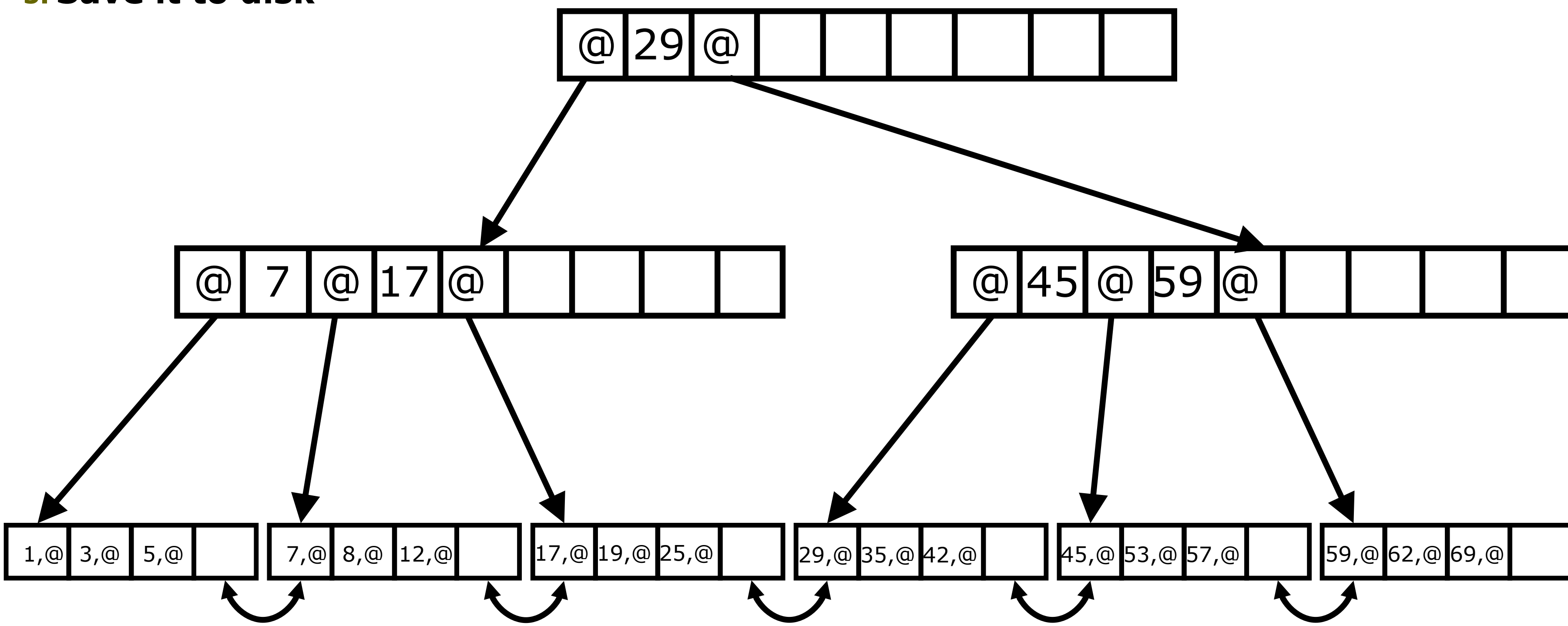


[1,@],[3,@],[5,@],[7,@],[8,@],[12,@],[17,@],[19,@],[25,@],[29,@],[35,@],[42,@],[45,@],[53,@],[57,@],[59,@],[62,@],[69,@]

Example of building a B-tree

1. **Create a file with entries [value,RID]**
2. **Sort the file by value**
3. **Build the leaves (filling them to the desired load)**
4. **Build the internal nodes (filling them to the desired load)**
5. **Save it to disk**

Order = 2
Load = 75%



[1,@],[3,@],[5,@],[7,@],[8,@],[12,@],[17,@],[19,@],[25,@],[29,@],[35,@],[42,@],[45,@],[53,@],[57,@],[59,@],[62,@],[69,@]

Comparison

A. Sequence of individual insertions
   1. For each new tuple
      1. Insert it -
      2. Find its place
      3. Write the leaf (Possible split)

B. Rebuild an index
   1. Remove the index -
   2. Insert new tuples -
   3. Read table (with new tuples)
   4. Write entries file
   5. Sort entries
   6. Write index

1. Copy leaves to entries file
2. Remove the index -
3. Insert new tuples and entries
4. Sort entries
5. Write index

We should define an index …

- B-tree:
  - There is a very selective condition
- Hash:
  - There is a very selective condition (with equality)
  - The table is not very volatile
  - The table is huge
- Clustered:
  - There is a little selective condition, or a GROUP BY or an ORDER BY or …
  - The table is not very volatile

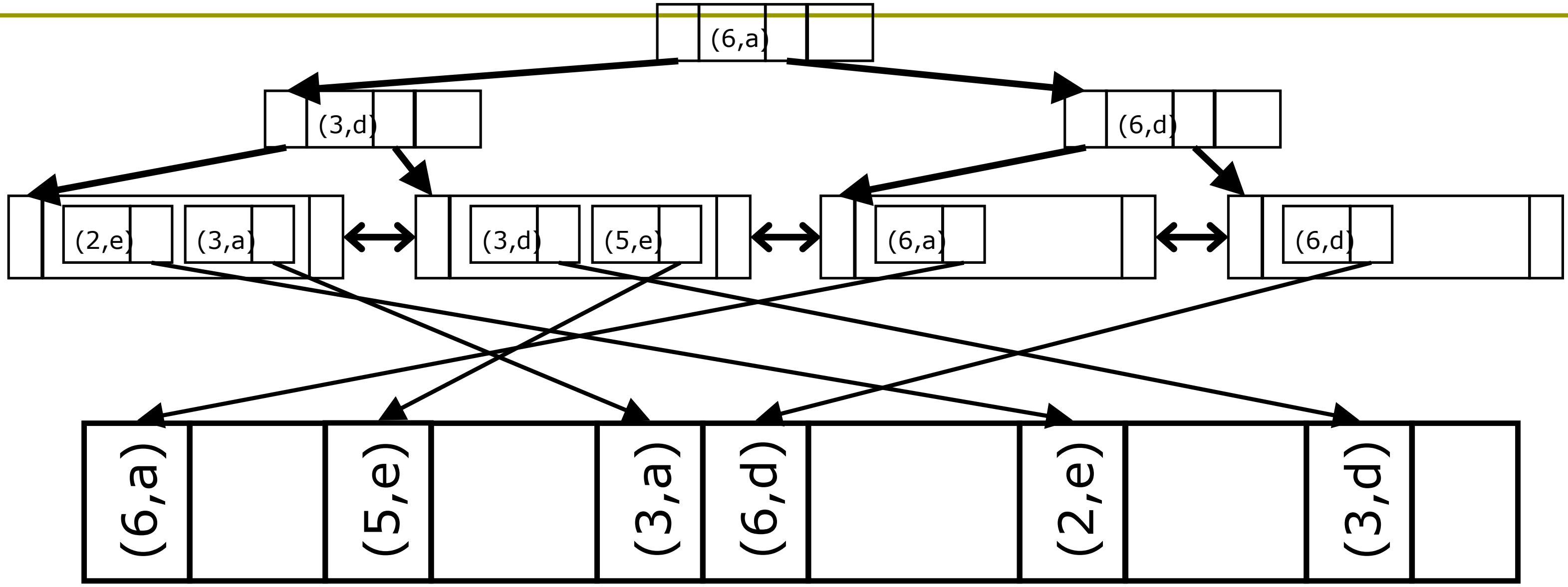
We should NOT define an index …

- Processing is massive
  - Never one tuple at a time
- The table has few blocks
- The attribute has few values
  - Little selective conditions
- The attribute appears in the predicate inside a function (maybe DBMS allows function-based indexes)

Usefulness of multi-attribute trees

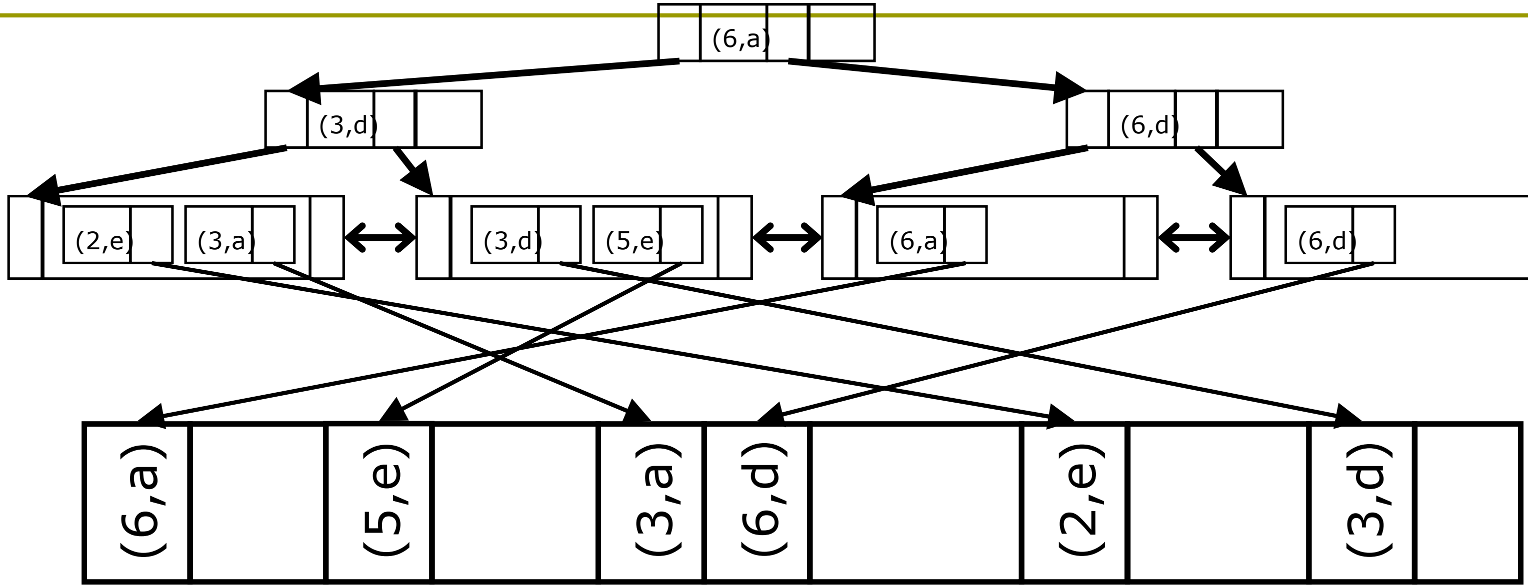
- Need more space
  - For each tuple, keeps attributes A_1, .., A_k
  - May result in more levels, worsening access time
- Modifications are more frequent
  - Every time one of the attributes in the index is modified
- It is much more efficient than intersecting RID lists (to evaluate conjunctions)
- Can be used to solve several kinds of queries
  - Equality of all first *i* attributes
  - Equality of all first *i* attributes and range of *i+1*
- The order of attributes in the index matters
  - We cannot evaluate condition over A_k, if there is no equality for A_1, .., A_{k-1}

Multi-attribute tree



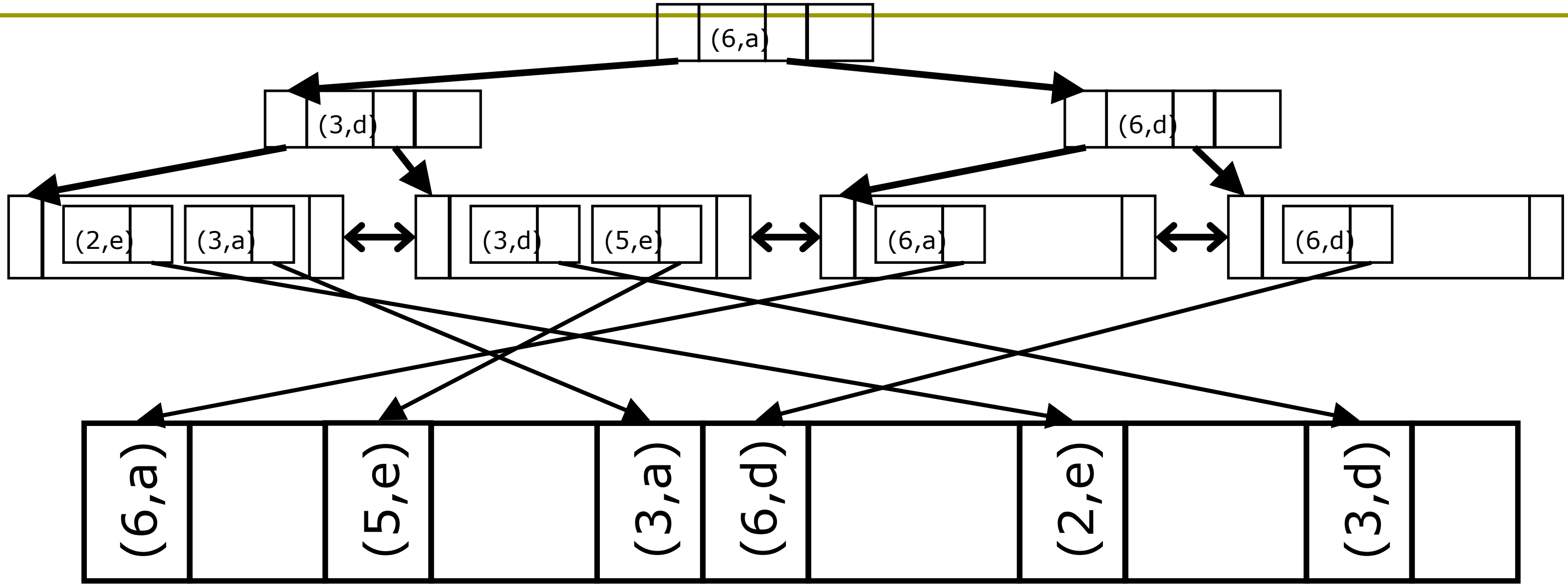
- Queries:
  - Num=‘3’ AND Let=‘d’
  - Num=‘3’ AND Let>‘b’
  - Num=‘3’
  - Num>’3’ AND Let=‘a’
  - Num>’3’ AND Let>‘b’
  - Num>’3’
  - Let=‘e’
  - Let>’b’
  - Num=‘3’ OR Let=‘a’

Multi-attribute tree



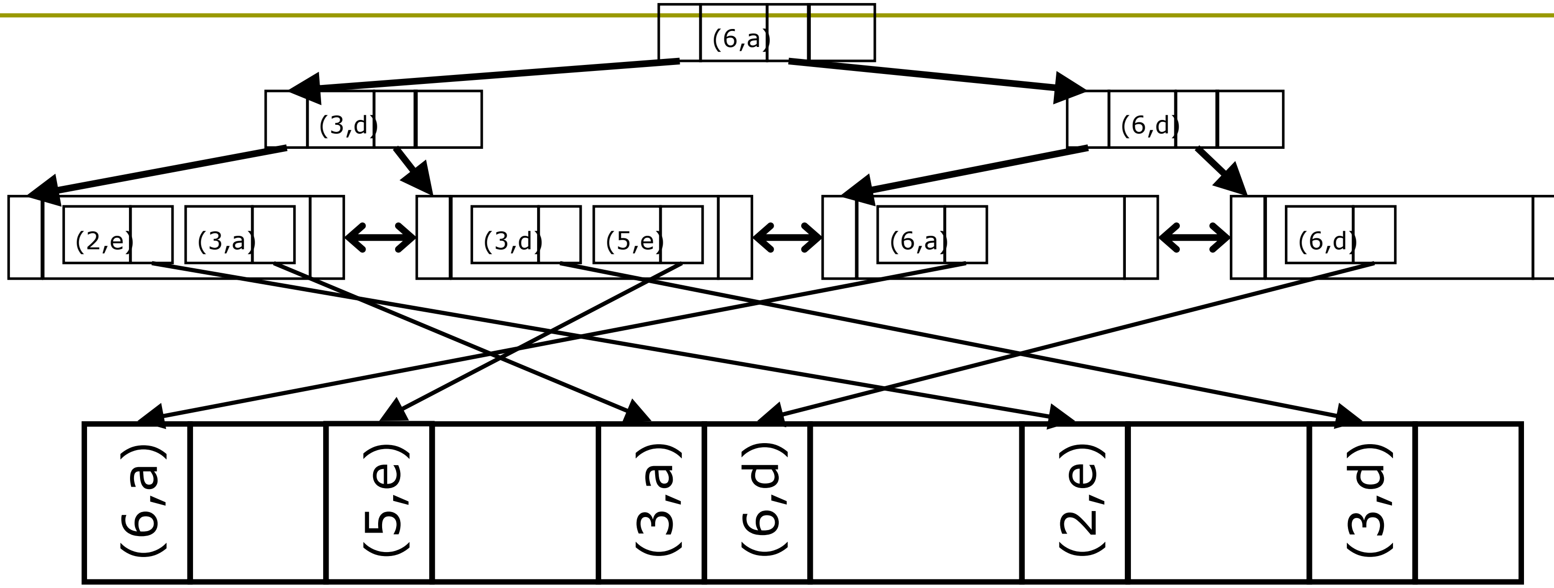
- Queries:
  - Num=‘3’ AND Let=‘d’ YES
  - Num=‘3’ AND Let>‘b’
  - Num=‘3’
  - Num>’3’ AND Let=‘a’
  - Num>’3’ AND Let>‘b’
  - Num>’3’
  - Let=‘e’
  - Let>’b’
  - Num=‘3’ OR Let=‘a’

Multi-attribute tree



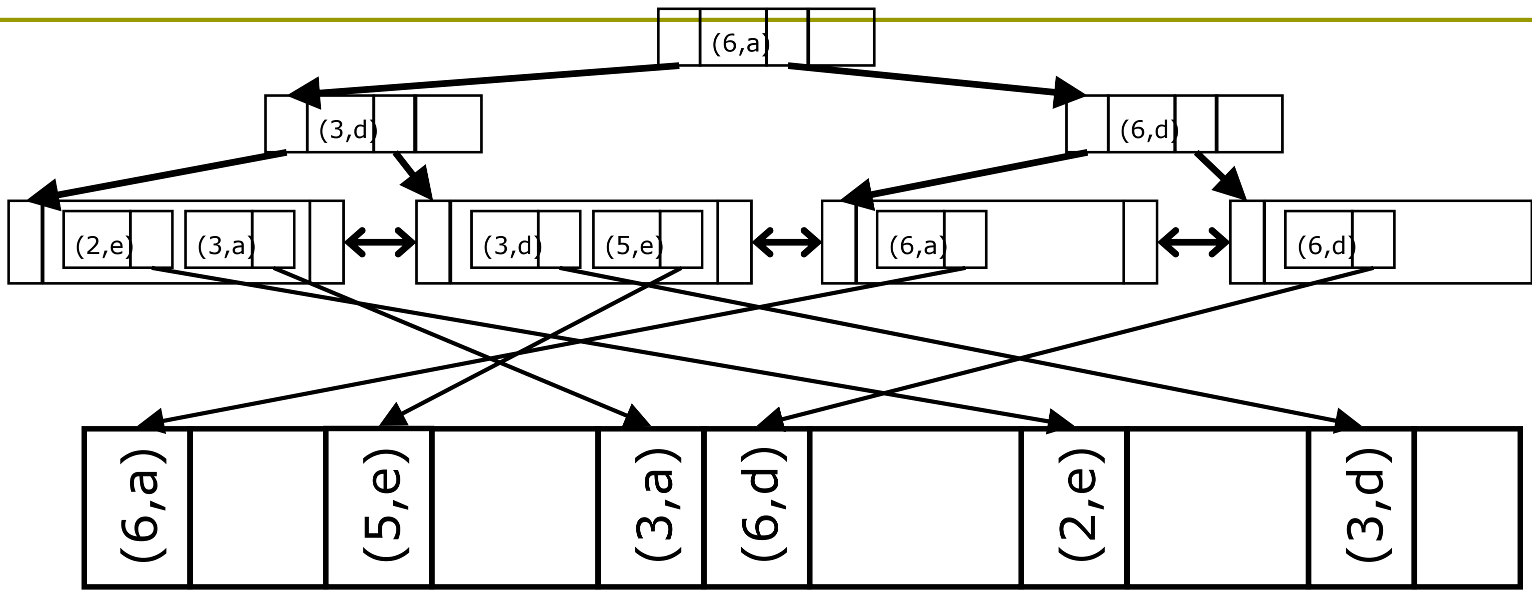
- Queries:
  - Num='3' AND Let='d' YES
  - Num='3' AND Let>'b' YES
  - Num='3'
  - Num>'3' AND Let='a'
  - Num>'3' AND Let>'b'
  - Num>'3'
  - Let='e'
  - Let>'b'
  - Num='3' OR Let='a'

Multi-attribute tree



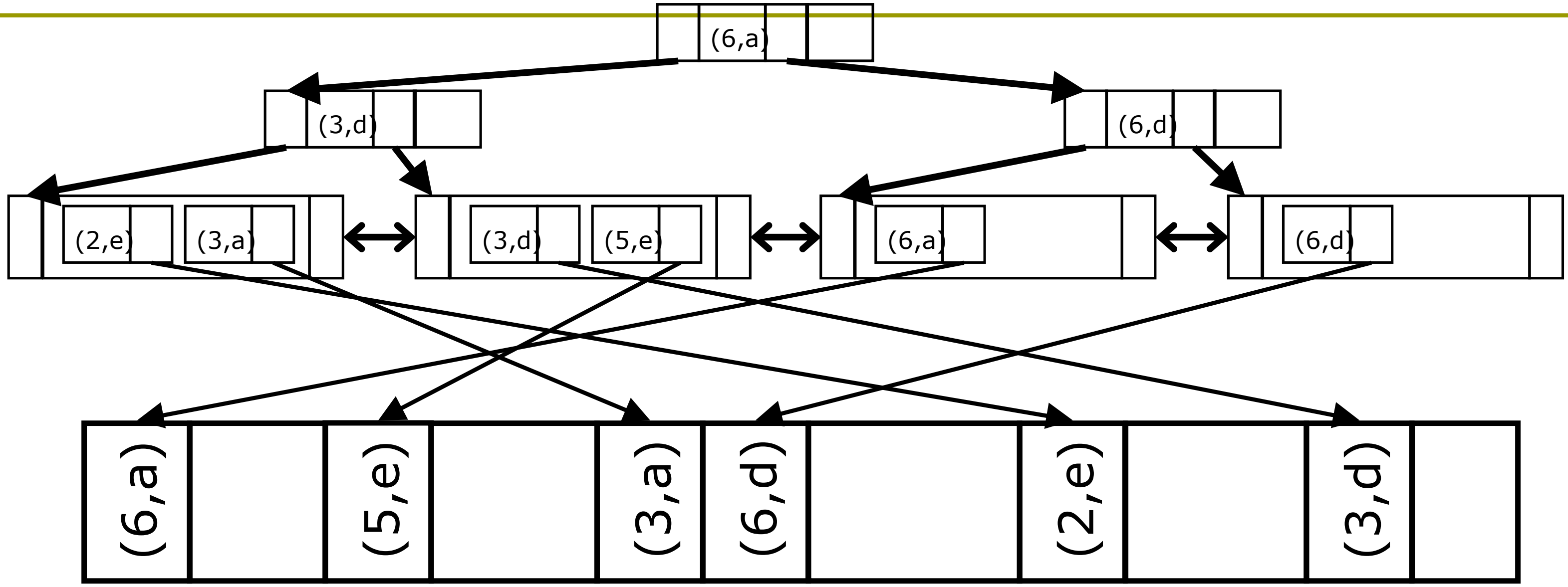
- Queries:
  - Num=‘3’ AND Let=‘d’ YES
  - Num=‘3’ AND Let>‘b’ YES
  - Num=‘3’ YES
  - Num>’3’ AND Let=‘a’
  - Num>’3’ AND Let>‘b’
  - Num>’3’
  - Let=‘e’
  - Let>’b’
  - Num=‘3’ OR Let=‘a’

Multi-attribute tree



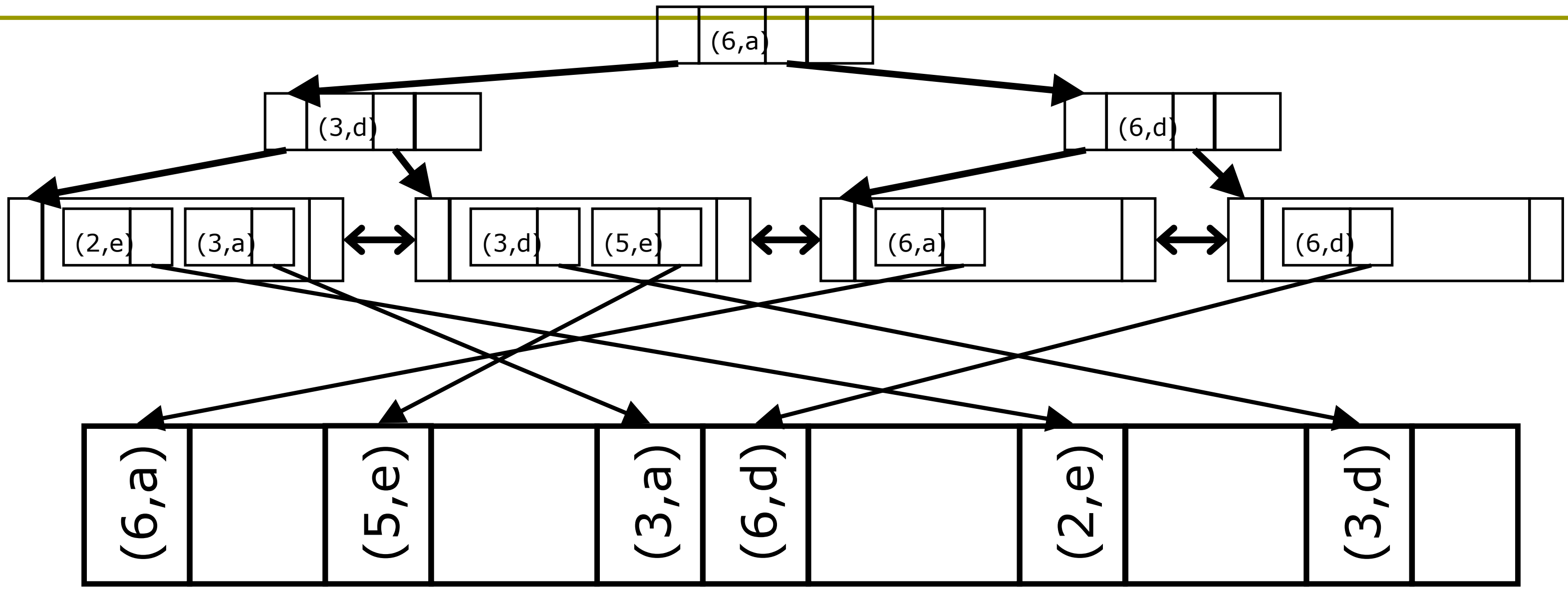
- Queries:
  - Num='3' AND Let='d' YES
  - Num='3' AND Let>'b' YES
  - Num='3' YES
  - Num>'3' AND Let='a' NO
  - Num>'3' AND Let>'b'
  - Num>'3'
  - Let='e'
  - Let>'b'
  - Num='3' OR Let='a'

Multi-attribute tree



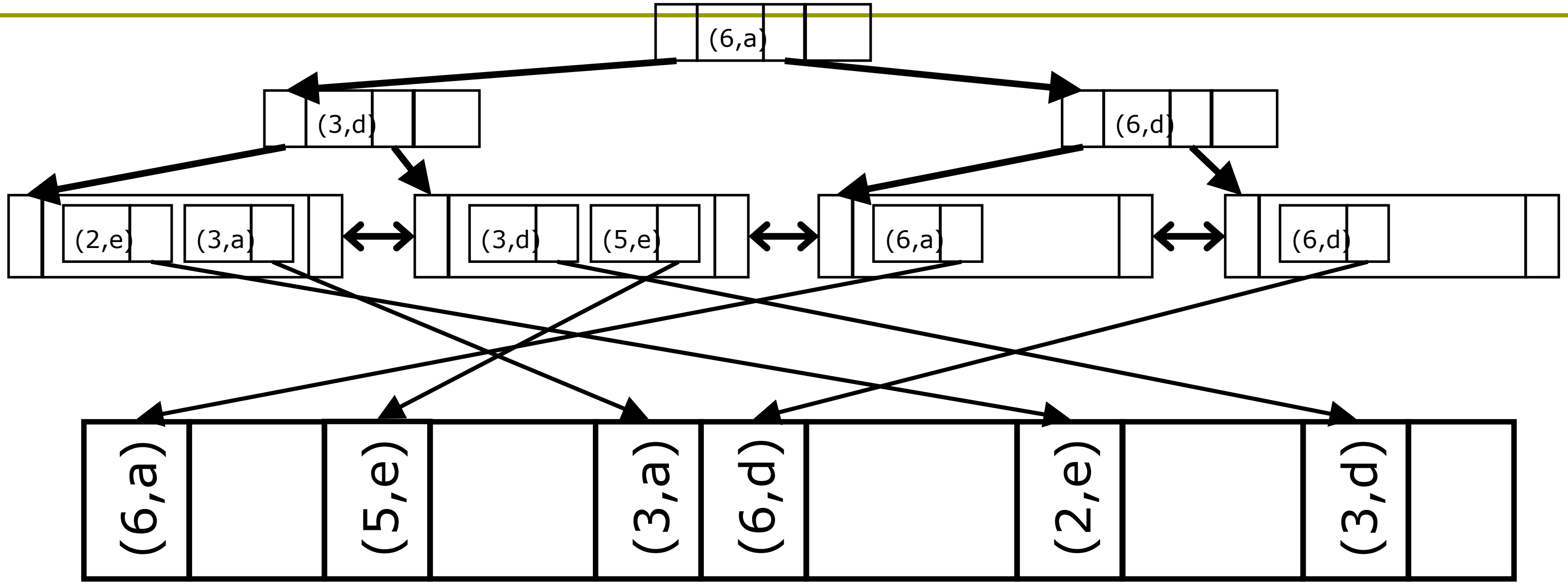
- Queries:
  - Num='3' AND Let='d' YES
  - Num='3' AND Let>'b' YES
  - Num='3' YES
  - Num>'3' AND Let='a' NO
  - Num>'3' AND Let>'b' NO
  - Num>'3'
  - Let='e'
  - Let>'b'
  - Num='3' OR Let='a'

Multi-attribute tree



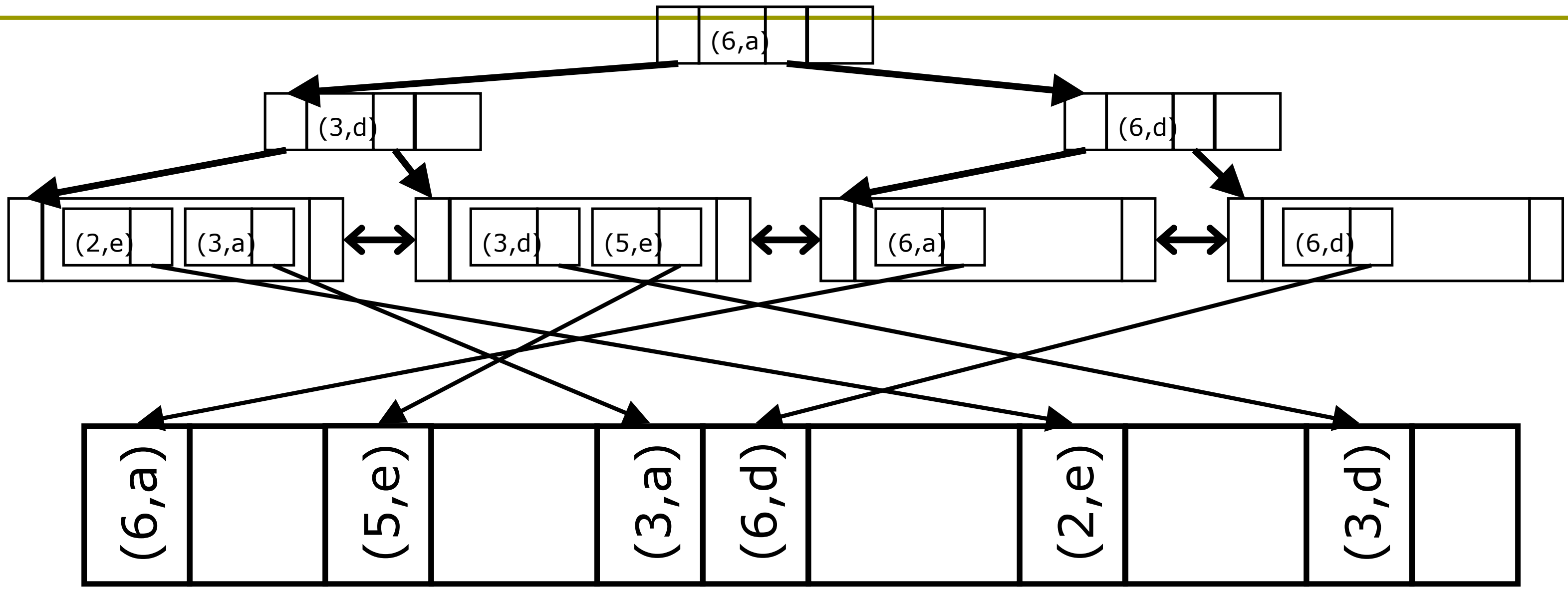
- Queries:
  - Num='3' AND Let='d' YES
  - Num='3' AND Let>'b' YES
  - Num='3' YES
  - Num>'3' AND Let='a' NO
  - Num>'3' AND Let>'b' NO
  - Num>'3' YES
  - Let='e'
  - Let>'b'
  - Num='3' OR Let='a'

Multi-attribute tree



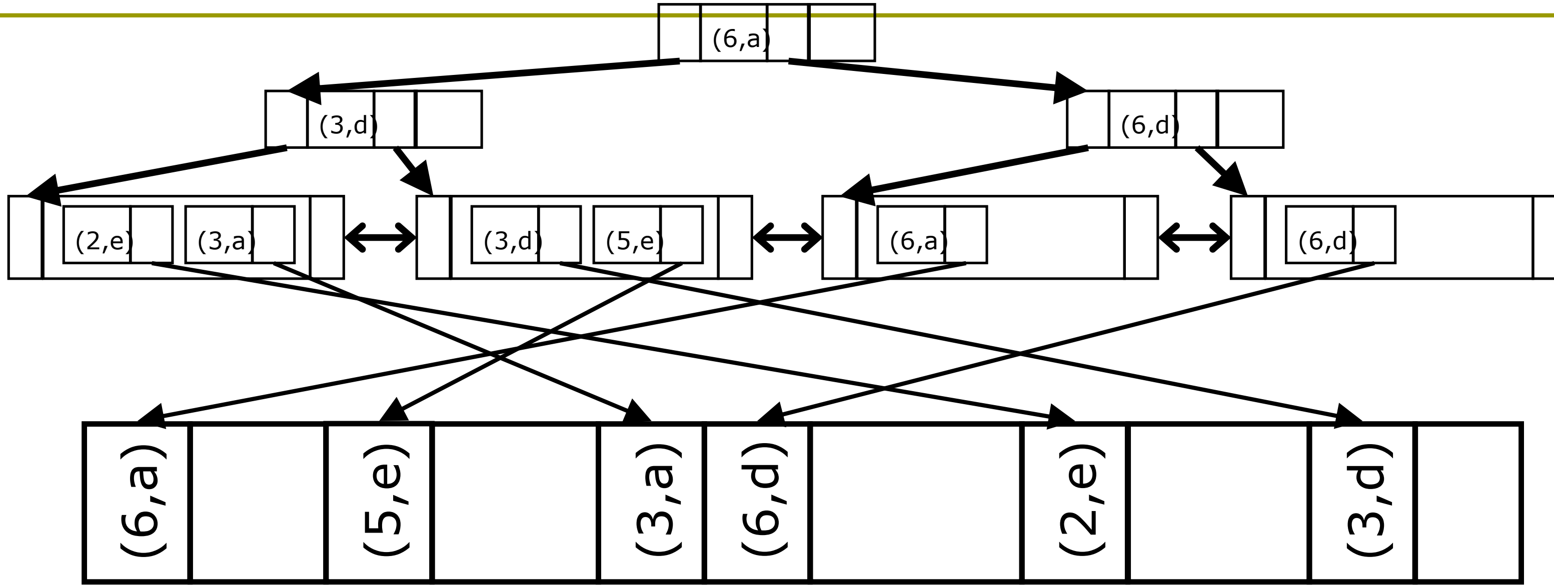
- Queries:
  - Num='3' AND Let='d' YES
  - Num='3' AND Let>'b' YES
  - Num='3' YES
  - Num>'3' AND Let='a' NO
  - Num>'3' AND Let>'b' NO
  - Num>'3' YES
  - Let='e' NO
  - Let>'b'
  - Num='3' OR Let='a'

Multi-attribute tree



- Queries:
  - Num=‘3’ AND Let=‘d’ YES
  - Num=‘3’ AND Let>‘b’ YES
  - Num=‘3’ YES
  - Num>’3’ AND Let=‘a’ NO
  - Num>’3’ AND Let>‘b’ NO
  - Num>’3’ YES
  - Let=‘e’ NO
  - Let>’b’ NO
  - Num=‘3’ OR Let=‘a’

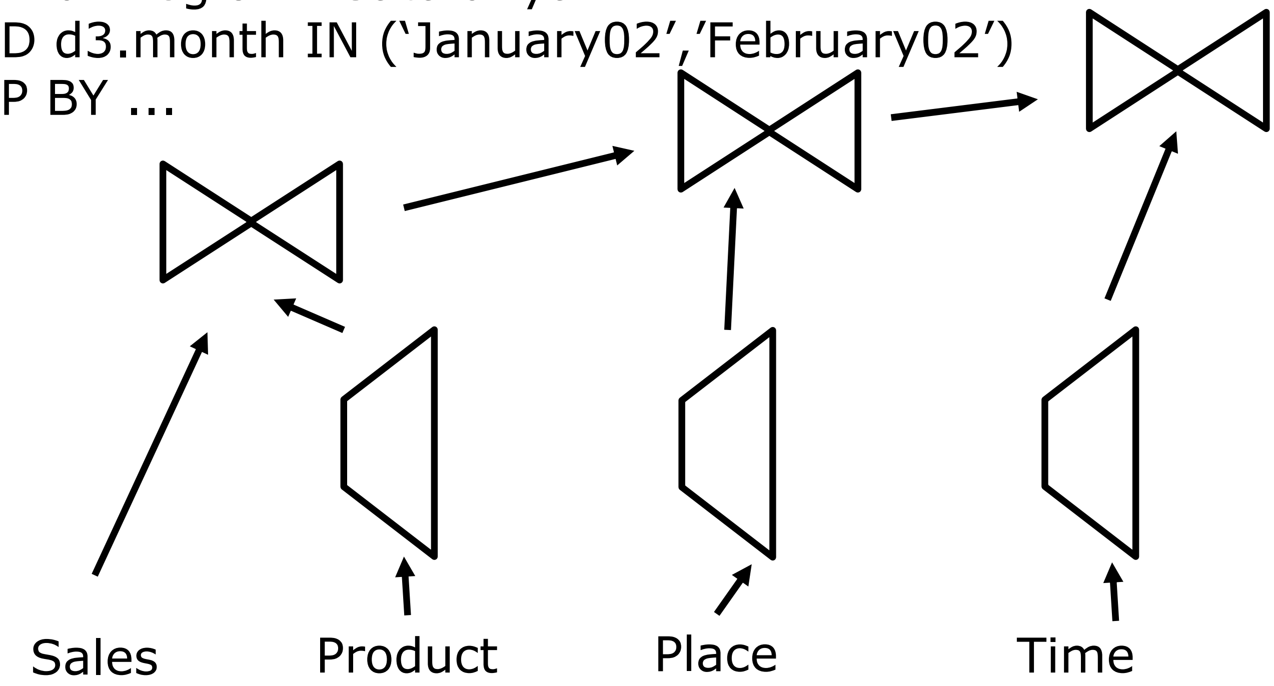
Multi-attribute tree



- Queries:
  - Num='3' AND Let='d' YES
  - Num='3' AND Let>'b' YES
  - Num='3' YES
  - Num>'3' AND Let='a' NO
  - Num>'3' AND Let>'b' NO
  - Num>'3' YES
  - Let='e' NO
  - Let>'b' NO
  - Num='3' OR Let='a' NO

Multidimensional queries

```
SELECT d1.articleName, d2.region, d3.month, SUM(f.articles)
FROM Sales f, Product d1, Place d2, Time d3
WHERE f.productId=d1.ID AND f.placeId=d2.ID AND f.timeId=d3.ID
    AND d1.articleName IN ('Ballpoint','Rubber')
    AND d2.region='Catalunya'
    AND d3.month IN ('January02','February02')
GROUP BY ...
```



Join-index (I)

Place(	placeId,	city,	region)
	111	BCN	Cat
	222	BIO	PV
	…		

Sales(	placeId,	timeId,…)
	111	T1
	222	T2
	222	T3
	111	T4
	…	

Time(	timeId,	year)
	T1	2000
	T2	2000
	T3	2001
	T4	2001
	…	

Join-index (I)

Place(	placeId,	city,	region)
	111	BCN	Cat
	222	BIO	PV
	…		

Sales(	placeId,	timeId,…)
	111	T1
	222	T2
	222	T3
	111	T4
	…	

Time(	timeId,	year)
	T1	2000
	T2	2000
	T3	2001
	T4	2001
	…	

Join-index (one attribute)

Cat	1,4, …
PV	2,3, …

Join-index (I)

Place(	placeId,	city,	region)
	111	BCN	Cat
	222	BIO	PV
	…		

Sales(	placeId,	timeId,…)
	111	T1
	222	T2
	222	T3
	111	T4
	…	

Time(	timeId,	year)
	T1	2000
	T2	2000
	T3	2001
	T4	2001
	…	

Join-index (one attribute)

Cat	1,4, …
PV	2,3, …

Join-index (two attributes)

Cat	2000	1, …
	2001	4, …
PV	2000	2, …
	2001	3, …

Join-index (I)

Place(	placeId,	city,	region)
	111	BCN	Cat
	222	BIO	PV
	…		

Sales(	placeId,	timeId,…)
	111	T1
	222	T2
	222	T3
	111	T4
	…	

Time(	timeId,	year)
	T1	2000
	T2	2000
	T3	2001
	T4	2001
	…	

Join-index (one attribute)

Cat	1,4, …
PV	2,3, …

Join-index (two attributes)

Cat	2000	1, …
	2001	4, …
PV	2000	2, …
	2001	3, …

Join-index (two attributes)

2000	Cat	1, …
	PV	2, …
2001	Cat	4, …
	PV	3, …

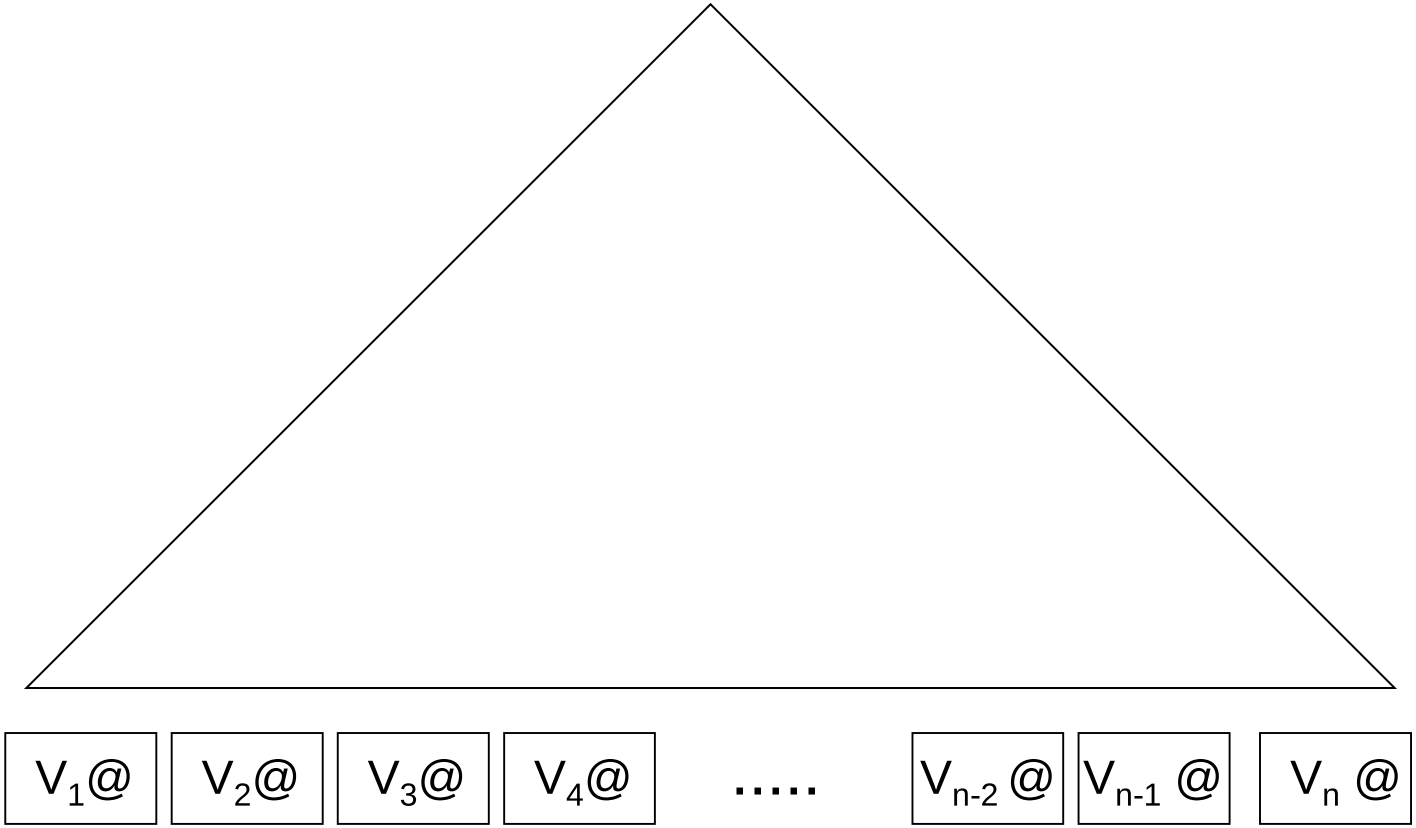
Join-index (II)

- The algorithm is the one inside the loop in Row Nested Loops
- The saving in multidimensional queries is
  - Dimension tables do not need to be accessed
- Considerations
  - It is really useful if there is a join-index over the selection attribute of R
  - It may be useful even if there is a join-index over the join attribute of R
  - A hash index can only be used for equi-join
  - We cannot use this algorithm if we already performed another operation over the table (we could decide to change the syntactic tree)

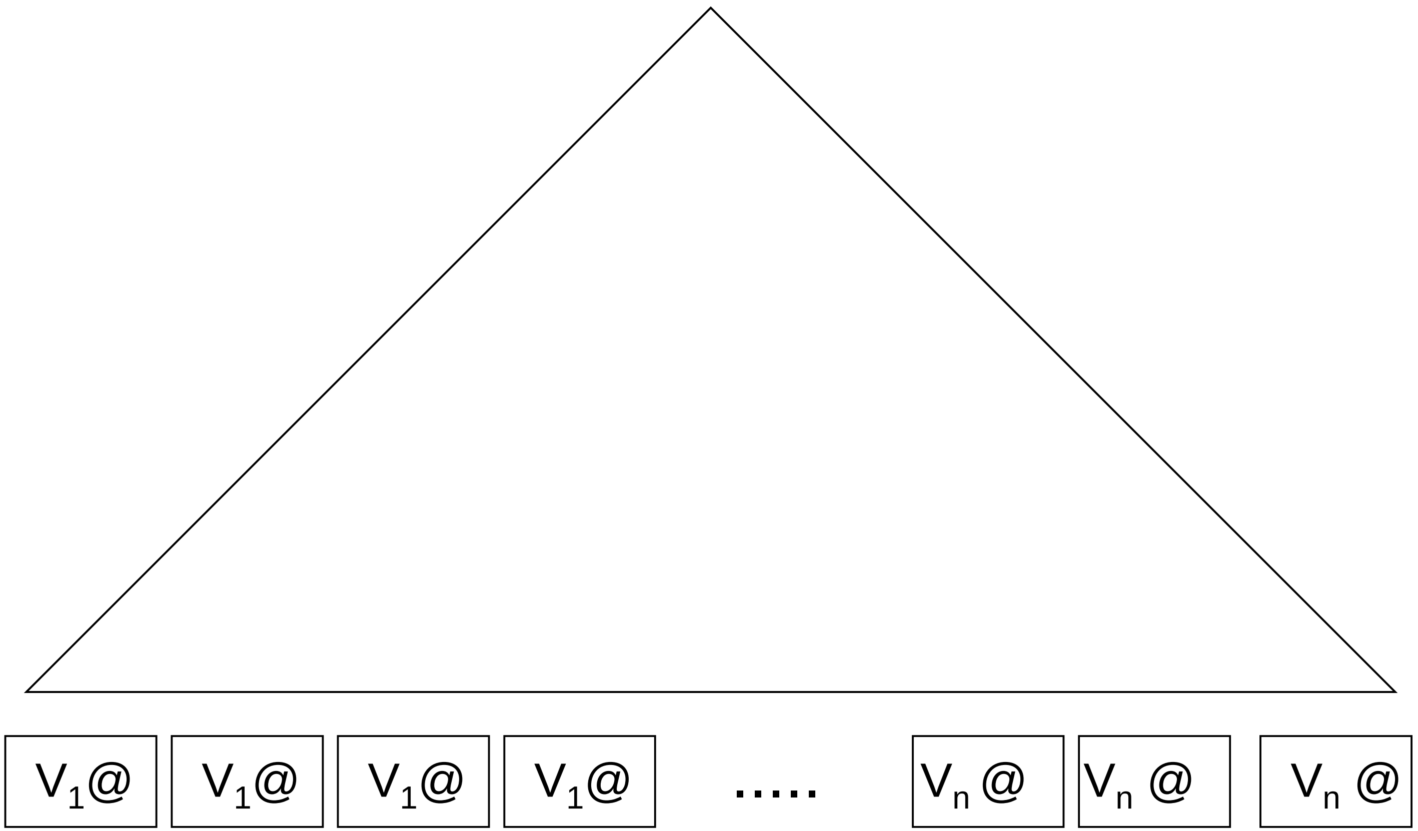
B-tree (I)

- Specially useful for simple queries
  - Without grouping, aggregations, or many joins
- Works better for very selective attributes (few repetitions per value)
  - Attributes in multidimensional queries are usually not very selective
- Order of attributes in the index is relevant
  - We can define as many indexes as we want
    - We can define only one Clustered index
    - For big tables, they may use too much space

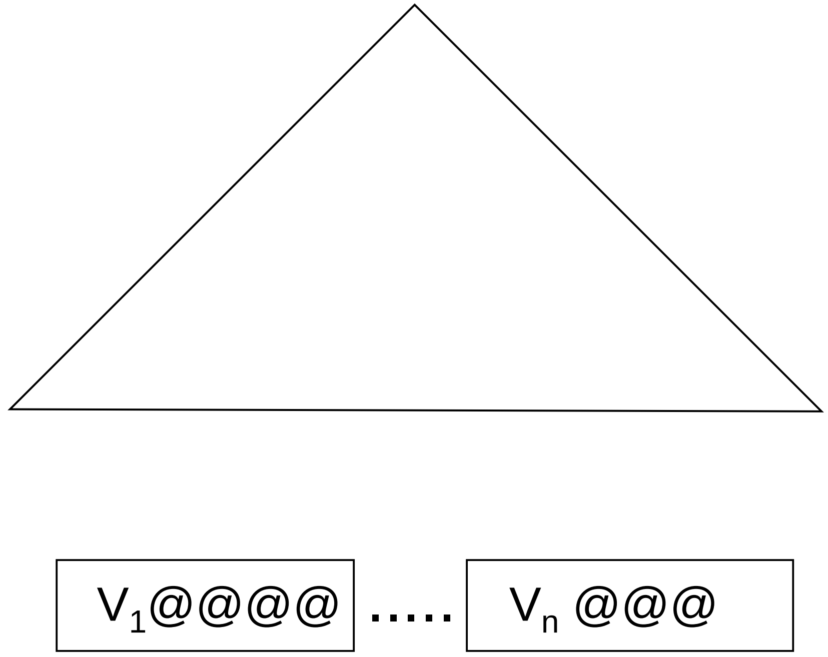
B-tree (II)



B-tree (II)



B-tree (II)



B-tree (II)

V_1@@...	V_2 @@...	V_3 @@...	V_4 @@...

Bitmap-index

Ballpoint	Pencil	Pen	Rubber	A4 paper	A3 paper	Chalk	Eraser
1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	1
0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0
0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0

Catalunya	León	Madrid	Andalucía
1	0	0	0
1	0	0	0
0	0	0	1
0	0	1	0
0	1	0	0
1	0	0	0
0	0	0	1
0	1	0	0
1	0	0	0
1	0	0	0

Querying with bitmaps

SELECT COUNT(*)

…

WHERE articleName IN [‘Ballpoint’,‘Pencil’] AND region=‘Catalunya’

Ballpoint		Pencil				Catalunya		
1		0		1		1		1
0		0		0		1		0
0		1		1		0		0
0		0		0		0		0
0	OR	0		0	AND	0		0
1		0	=	1		1	=	1
0		0		0		0		0
0		0		0		0		0
0		0		0		1		0
0		1		1		1		1

Updating bitmaps

- Two cases of insertion:
  - Without domain expansion:
    - Add "1"
  - With domain expansion:
    - Add a new vector
- One case of deletion:
    - Change "1" for "0"

Catalunya	León	Madrid	Andalucía
1	0	0	0
1	0	0	0
0	0	0	1
0	0	1	0
0	1	0	0
1	0	0	0
0	0	0	1
0	1	0	0
1	0	0	0
1	0	0	0

Updating bitmaps

- Two cases of insertion:
  - Without domain expansion:
    - Add “1”
  - With domain expansion:
    - Add a new vector
- One case of deletion:
    - Change “1” for “0”

Catalunya	León	Madrid	Andalucía
1	0	0	0
1	0	0	0
0	0	0	1
0	0	1	0
0	1	0	0
1	0	0	0
0	0	0	1
0	1	0	0
1	0	0	0
1	0	0	0
0	0	1	0

Updating bitmaps

- Two cases of insertion:
  - Without domain expansion:
    - Add “1”
  - With domain expansion:
    - Add a new vector

- One case of deletion:
    - Change “1” for “0”

Catalunya	León	Madrid	Andalucía	Euskadi
1	0	0	0	0
1	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	1	0	0	0
1	0	0	0	0
0	0	0	1	0
0	1	0	0	0
1	0	0	0	0
1	0	0	0	0
0	0	1	0	0
0	0	0	0	1

Updating bitmaps

- Two cases of insertion:
  - Without domain expansion:
    - Add "1"
  - With domain expansion:
    - Add a new vector

- One case of deletion:
    - Change "1" for "0"

Catalunya	León	Madrid	Andalucía	Euskadi
0	0	0	0	0
1	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	1	0	0	0
1	0	0	0	0
0	0	0	1	0
0	1	0	0	0
1	0	0	0	0
1	0	0	0	0
0	0	1	0	0
0	0	0	0	1

Probabilities with a bitmap

- Probability of a tuple fulfilling P
  SF
- Probability of a tuple NOT fulfilling P
  1-SF
- Probability of none of the tuples in a block fulfilling P
  $(1-SF)\cdot(1-SF)\cdot \ldots \cdot(1-SF) = (1-SF)^R$
- Probability of some tuple in a block fulfilling P
  $1-(1-SF)^R$

Cost of bitmap per operation

bits: bits per index block
ndist: different values
v: number of queried values

- Table scan
  - $ndist \cdot \lceil |T|/bits \rceil \cdot D + B \cdot D$
- Search for some tuples
  - $v \cdot \lceil |T|/bits \rceil + (B \cdot (1-(1-SF)^R))$
  - Examples:
    - Search for one tuple
      - $\lceil |T|/bits \rceil \cdot D + D$
    - Search for several tuples (given one value)
      - $\lceil |T|/bits \rceil \cdot D + (B \cdot (1-((ndist-1)/ndist)^R))$
    - Search for several tuples (given several values)
      - $v \cdot \lceil |T|/bits \rceil \cdot D + (B \cdot (1-((ndist-v)/ndist)^R))$
- Insertion of one tuple (in the last table block)
  - Existing value: $ndist \cdot 2 + 2$
  - New value: $ndist \cdot 2 + 2 + \lceil |T|/bits \rceil$
- Deletion of all tuples with a given value
  - $\lceil |T|/bits \rceil + (B \cdot (1-((ndist-1)/ndist)^R)) \cdot 2$

Cost of bitmap per operation

bits: bits per index block
ndist: different values
v: number of queried values

- Table scan
  - **Useless**
- Search for some tuples
  - $v \cdot \lceil |T|/bits \rceil + (B \cdot (1-(1-SF)^R))$
  - Examples:
    - Search for one tuple
      - $\lceil |T|/bits \rceil \cdot D + D$
    - Search for several tuples (given one value)
      - $\lceil |T|/bits \rceil \cdot D + (B \cdot (1-((ndist-1)/ndist)^R))$
    - Search for several tuples (given several values)
      - $v \cdot \lceil |T|/bits \rceil \cdot D + (B \cdot (1-((ndist-v)/ndist)^R))$
- Insertion of one tuple (in the last table block)
  - Existing value: $ndist \cdot 2 + 2$
  - New value: $ndist \cdot 2 + 2 + \lceil |T|/bits \rceil$
- Deletion of all tuples with a given value
  - $\lceil |T|/bits \rceil + (B \cdot (1-((ndist-1)/ndist)^R)) \cdot 2$

Cost of bitmap per operation

bits: bits per index block
ndist: different values
v: number of queried values

- Table scan
  - **Useless**
- Search for some tuples
  - $v \cdot \lceil |T|/bits \rceil + (B \cdot (1-(1-SF)^R))$
  - Examples:
    - Search for one tuple
      - **Useless?**
    - Search for several tuples (given one value)
      - $\lceil |T|/bits \rceil \cdot D + (B \cdot (1-((ndist-1)/ndist)^R))$
    - Search for several tuples (given several values)
      - $v \cdot \lceil |T|/bits \rceil \cdot D + (B \cdot (1-((ndist-v)/ndist)^R))$
- Insertion of one tuple (in the last table block)
  - Existing value: $ndist \cdot 2+2$
  - New value: $ndist \cdot 2+2+\lceil |T|/bits \rceil$
- Deletion of all tuples with a given value
  - $\lceil |T|/bits \rceil + (B \cdot (1-((ndist-1)/ndist)^R)) \cdot 2$

Comparison

- Better than B-tree and hash for multi-value queries
- Optimum performance for several conditions over more than one attribute (each with a low selectivity)
- Orders of magnitude of improvement compared to a table scan (specially for SF<1%)
- May be useful even for range queries
- Easy indexing of NULL values
- Useful for non-unique attributes (specially for ndist<|T|/100, i.e. hundreds of repetitions)
- Bad performance for concurrent INSERT, UPDATE and DELETE
- Use more space than RID lists for domains of 32 values or more (may be better with compression), assuming uniform distribution and 4 bytes per RID

Bitmap indexes in Oracle 11g

```
CREATE
[{UNIQUE|BITMAP}] INDEX <name>
ON <table> (<column>[,column]*);
```

- Allowed even for unique attributes
- Does not allow to check uniqueness

Benefits of Bitmap-join-index

```
CREATE BITMAP INDEX salesRegion
ON Sales(Place.region)
FROM Sales, Place
WHERE Sales.placeId=Place.ID;
```

- The saving in multidimensional queries is
  - Bit operations substitute unions and intersections
- Comparison against Clustered Structure:
  - Space: Always better
  - Time: Better for several relatively selective conditions

Index-only query answering

- Projection

  SELECT age
  FROM people

  SELECT DISTINCT age
  FROM people

  - Attribute removal
    - B+
      - 1.5(|T|/2d)·D
    - Hash
      - 1.25(|T|/2d)·D
- Aggregates

  SELECT MIN(age)
  FROM people

  SELECT AVG(age)
  FROM people

  SELECT age, COUNT(*)
  FROM people
  GROUP BY age;
- Joins

  SELECT p.name
  FROM people p, departaments d
  WHERE p.id=d.boss;

  - Index Sort Match Join
  - Index Block Nested Loops
  - Index Row Nested Loops

Summary

- Algorithm for B-tree rebuilding
- We should/shouldn’t define an index…
- Bitmap-index
- Multi-attribute index usage
- Join-index
- Index-only query answering

Bibliography

- D. Shasha and P. Bonnet. *Database Tunning*. Elsevier, 2003
- C. T. Yu and W. Meng. *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann, 1998
- P. Valduriez. *Join Indices*. ACM TODS, 12 (2), June 1987. Pages 218-246
- R. Ramakrishnan and J. Gehrke. *Database Management Systems*. 3rd edition. McGraw-Hill, 2003
- M. Golfarelli and S. Rizzi. *Data Warehouse Design*. McGrau-Hill, 2009