Part XIII

Index Support

Outline of this part

Index support

All known database indexing techniques (such as B^+ trees, hashing, . . .) can be employed to—depending on the chosen representation—support some or all of the following:

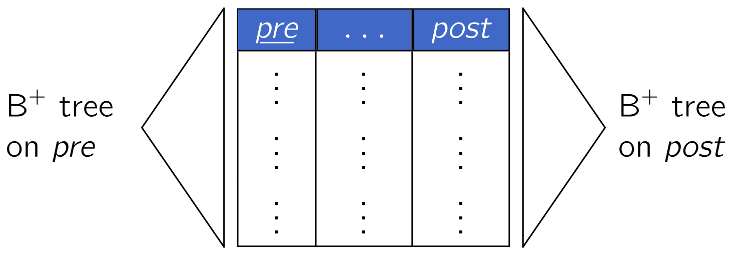
- uniqueness of node IDs,
- direct access to a node, given its node ID,
- ordered sequential access to document parts (serialization),
- name tests,
- value predicates,
- structural traversal along some or all of the XPath axes,
- . . .

We will only look into a few interesting special cases here.

Pre/*post* encoding and B^+ trees

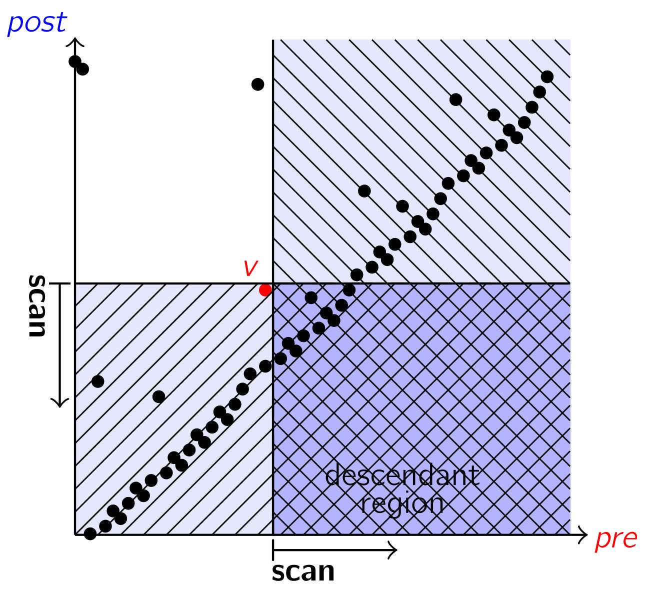
As we have already seen before, the XPath Accelerator encoding leads to *conjunctions* of a lot of *range selection predicates* on the *pre* and *post* attributes in the resulting SQL queries.

Two B^+ tree indexes on the *accel* table, defined over *pre* and *post* attributes:



Query evaluation (example)

Evaluating, *e.g.*, a `descendant` step can be supported by either one of the B^+ trees:



Two options:

① Use index on *pre*.

- Start at *v* and **scan** along *pre*.
- Many **false hits**!

② Use index on *post*.

- Start at *v* and **scan** along *post*.
- Many **false hits**!

- **Many false hits either way!**

Query evaluation using index intersection

Standard B^+ trees on those columns will support *really* efficient query evaluation, if the DBMS optimizer generates *index intersection* evaluation plans.

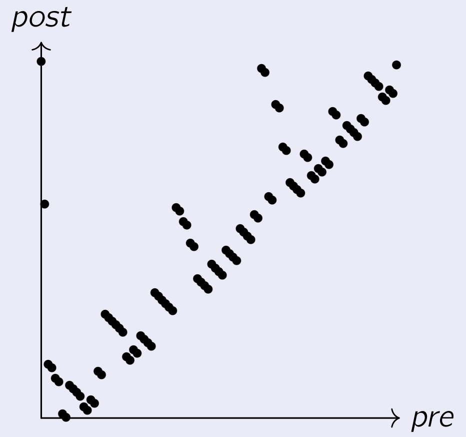
Query evaluation plans for predicates of the form "$pre \in [\ldots] \land post \in [\ldots]$" will then

1. evaluate both indexes separately to obtain pointer lists,
2. merge (*i.e.*, intersect) the pointer lists,
3. only *afterwards* access `accel` tuples satisfying *both* predicates.

Pre/post encoding and R trees

In the geometric/spatial database application area, quite a few *multi-dimensional* index structures have been developed. Such indexes support range predicates along arbitrary combinations of dimensions.

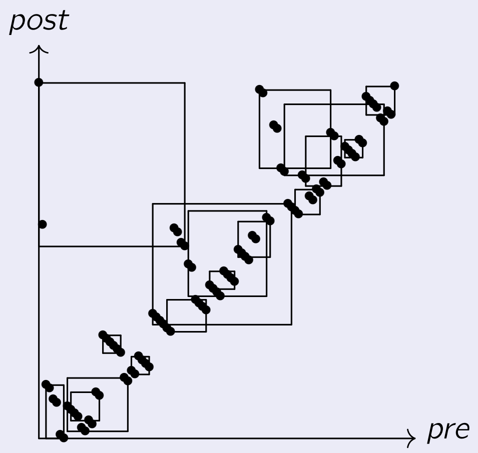
Pre/post encoding of a 100-node XML fragment



- Diagonal of *pre/post* plane densely populated.
- R-Trees partition plane incompletely, adapts well to node distribution.
- Node encodings are points in 5-dimensional space.
- 5-dimensional R-Tree evaluates XPath **axis** and **node tests** in **parallel.**

Preorder packed R tree

R tree loaded in ascending preorder, leaf capacity 6 nodes



- Insert node encodings into R tree in ascending order of *pre* ranks.
- Storage utilization in R tree leaf pages maximized.
- Coverage and overlapping of leaves minimized.
- Typical: preorder packing **preserves document order** on retrieval.

More on physical design issues

As always, chosing a clever physical database layout can greatly improve query (and update) performance.

- Note that all information necessary to evaluate XPath **axes** is encoded in columns *pre* and *post* (and *par*) of table *accel*.
- Also, **kind tests** rely on column *kind*, **name tests** on column *tag* only.

Which columns are required to evaluate the steps below?

Location step	**Columns needed**
`descendant::text()`	
`ancestor::x`	
`child::comment()`	
`/descendant-or-self::y`	

Splitting the encoding table

These observations suggest to split *accel* into **binary tables:**

Full split of *accel* table

prepost

pre	*post*
0	9
1	1
2	0
3	2
4	8
5	5
6	3
7	4
8	7
9	6

prepar

pre	*par*
0	NULL
1	0
2	1
3	0
4	0
5	4
6	5
7	5
8	4
9	8

prekind

pre	*kind*
0	elem
1	elem
2	text
3	com
4	elem
5	elem
6	elem
7	pi
8	elem
9	text

pretag

pre	*tag*
0	a
1	b
4	e
5	f
6	g
8	i

pretext

pre	*text*
2	c
3	d
7	h
9	j

- **NB.** Tuples are **narrow** (typically $\leqslant 8$ bytes wide)
  - ⇒ reduce amount of (secondary) memory fetched
  - ⇒ lots of tuples fit in the buffer pool/CPU data cache

“Vectorization”

- In an **ordered** storage (clustered index!), the *pre* column of table *prepost* is plain redundant.
- Tuples even narrower. Tree shape now encoded by ordered integer sequence (*cf.* “data vectors” idea).

Dense *pre* column

prepost

	post
	9
	1
	0
	2
	8
	5
	3
	4
	7
	6

- Use **positional access** to access such tables ($\rightarrow$ MonetDB).
  - Retrieving a tuple t in row $\#n$ implies $t.pre = n$.

Indexes on encoding tables?

- Analyse compiled XPath query to obtain advise on which **indexes** to create on the encoding tables.[42]

path(`fn:root()/descendant-or-self::a/descendant::text()`)

```
SELECT DISTINCT v1.pre
  FROM accel v2, accel v1
 WHERE v2.kind = elem and v2.tag = a        ::a
   AND v1.pre > v2.pre                      } descendant
   AND v1.post < v2.post
   AND v1.kind = text                       ::text()
ORDER BY v1.pre
```

[42]Supported by tools like the IBM DB2 *index advisor* `db2advis`.

Indexes on encoding tables

Query analysis suggests:

SQL index creation commands

```
1 CREATE        INDEX itag  ON accel (tag)
2 CREATE        INDEX ikind ON accel (kind)
3 CREATE        INDEX ipar  ON accel (par)
4 CREATE UNIQUE INDEX ipost ON accel (post ASC)
5 CREATE UNIQUE INDEX ipre  ON accel (pre ASC) CLUSTER
```

- ①–③: **Hash/B-tree indexes** ④–⑤: **B-tree indexes**

Resulting storage layout

Table and index contents (ordered!)

accel

RID	*pre*	*post*	⋯
ρ_0	0	9	
ρ_1	1	1	
ρ_2	2	0	
ρ_3	3	2	
ρ_4	4	8	
ρ_5	5	5	
ρ_6	6	3	
ρ_7	7	4	
ρ_8	8	7	
ρ_9	9	6	

ipost

RID	*post*
ρ_2	0
ρ_1	1
ρ_3	2
ρ_6	3
ρ_7	4
ρ_5	5
ρ_9	6
ρ_8	7
ρ_4	8
ρ_0	9

ikind

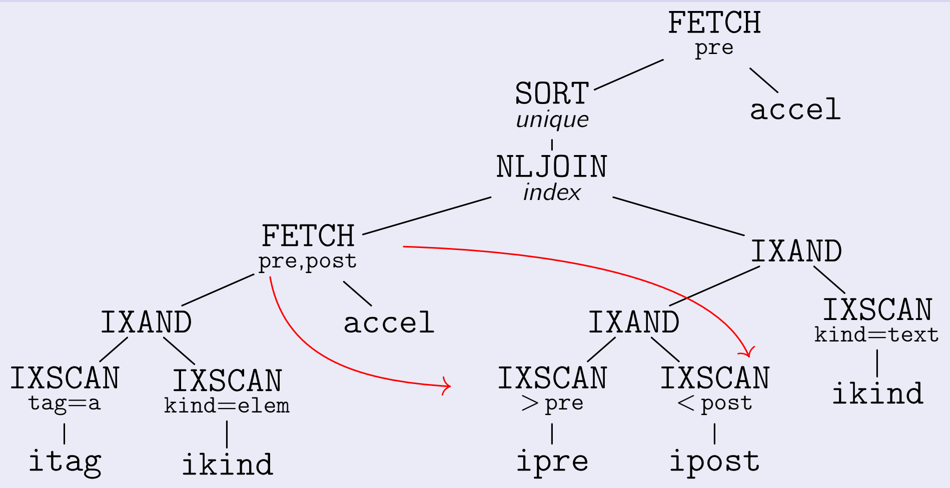
RID	*kind*
ρ_0	elem
ρ_1	elem
ρ_4	elem
ρ_5	elem
ρ_6	elem
ρ_8	elem
ρ_2	text
ρ_9	text
ρ_3	com
ρ_7	pi

Notes:

- ρ_i in `RID` column: database internal *row identifiers*.
- Rows of table `accel` ordered in preorder (`CLUSTER`).

Evaluation plan (DB2)

Plan for the query given above



A note on the IBM DB2 plan operators

Query plan operators used by IBM DB2 (excerpt)

Operator	**Effect**
`IXSCAN`	**Index scan** controlled by predicate on indexed column(s); yields row ID set
`IXAND`	**Intersection** of two row ID sets; yields row ID set
`FETCH`	Given a row ID set, **fetch specified columns** from table; yields tuple set
`SORT`	**Sort** given row ID/tuple set, optionally removing duplicates
`NLJOIN`	**Nested loops join**, optionally using index lookup for inner input
`TBSCAN`	**Scan entire table**, with an optional predicate filter