



Architecture Sensitive Design of Database Engines

Kenneth Ross
Columbia University

NEDS, November 19, 2004. Joint work with Jingren Zhou.

[Higher DBMS Performance]

Powerful High-Performance Processors

+

Compute, Memory-Intensive DB Apps

=

Higher Performance for DBMS?

- I/O is (usually) no longer the bottleneck

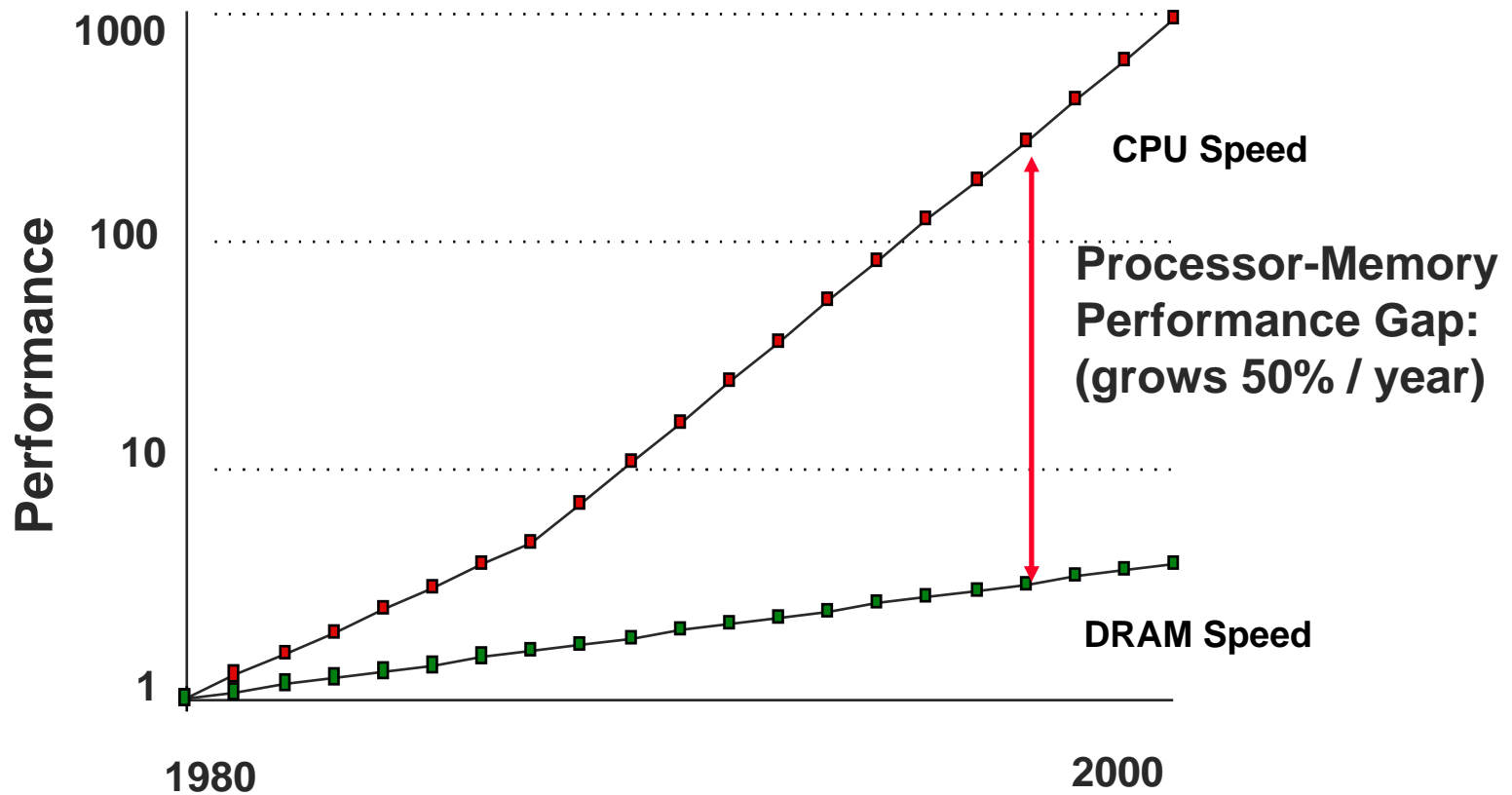
[Project Scope]

- Study (and improve) interactions between DBMS and processor/memory
 - Data cache conscious indexes
 - Improve instruction cache performance
 - Database operations using SIMD instructions
 - Optimize queries for branch misprediction behavior
 - Architecture-conscious data layout

[Today's Talk]

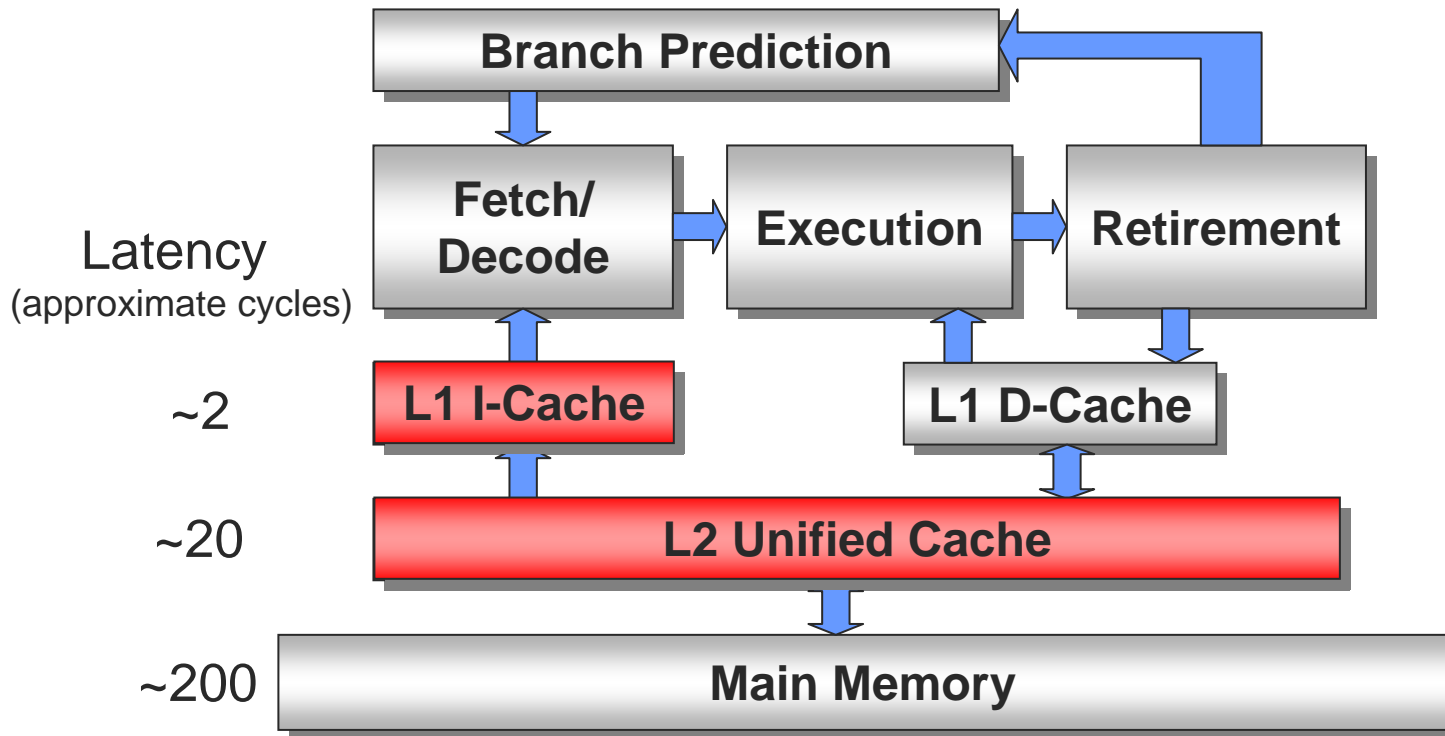
- Study (and improve) interactions between DBMS and processor/memory
 - Data cache conscious indexes
 - Improve instruction cache performance
 - Database operations using SIMD instructions
 - Optimize queries for branch misprediction behavior
 - Architecture-conscious data layout

Background: Processor and Memory Gap



(From Dave Patterson)

Background: Memory Hierarchy



Cache memories: designed based on the principle of *spatial and temporal locality*.

Background: Cache Optimization

- DBMS has no control of cache memories
- Previous cache optimization techniques
 - Clustering: packs successively accessed data together
 - Compression: removes irrelevant data
 - Coloring: maps data into non-conflicting cache regions
- Via **Buffering**: group things in time
 - ✓ Improves locality
 - ✓ Improves cache performance
 - ☹ Overhead
 - Maintaining buffering state information
 - Extra copy of elements?

[Outline]

- 1. Improve data cache performance**
 - **Buffering memory accesses to index structures**
- 2. Improve instruction cache performance**
 - Buffering intermediate results of a query subexpression

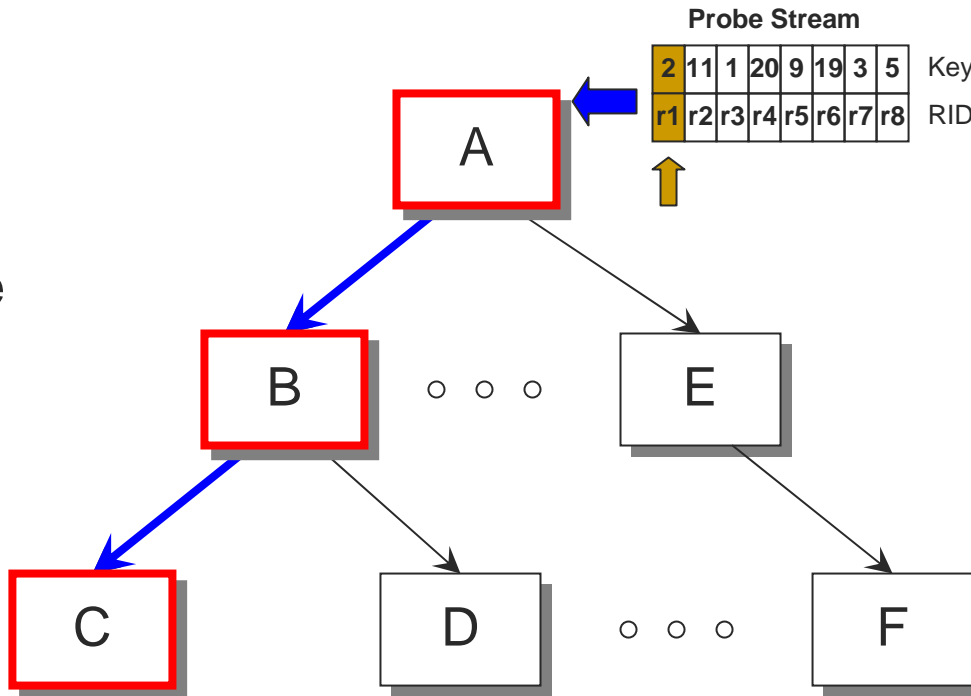
Cache-Conscious Indexes

- CSS-Tree, CSB⁺-Tree [Rao & Ross 2000], etc.
 - Use **compression**: eliminate all (or most) of the child pointers to increase the fanout of the trees
 - Node size matching the cache line size
- All focus on a **single** lookup
- **Many Consecutive** lookups?
 - Index-nested-loop
 - Stream data processing

Consecutive Lookups?



In Cache

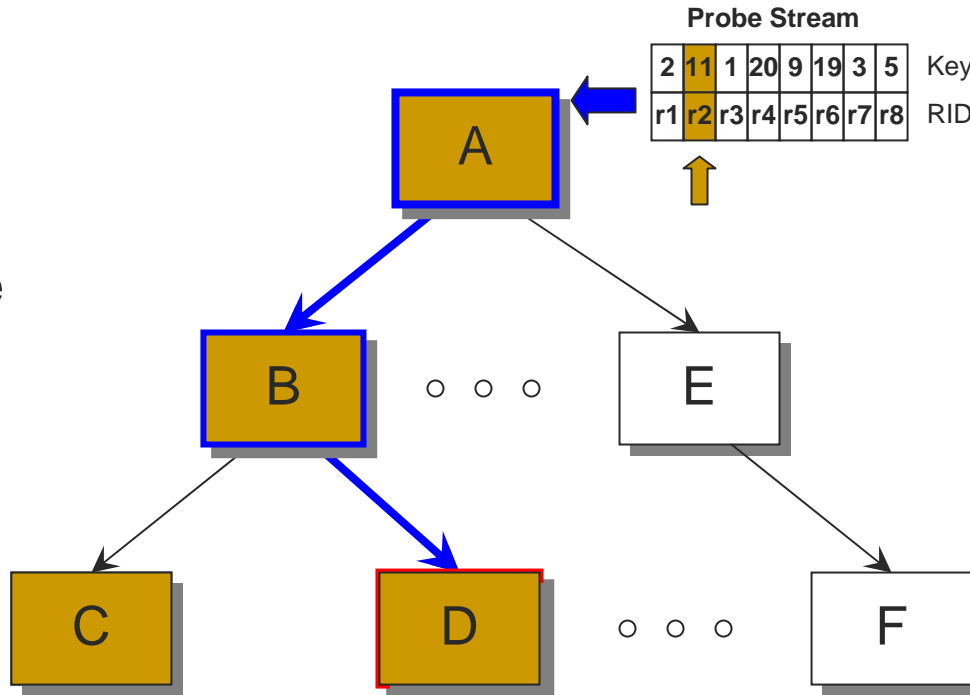


Cache Capacity
(Up to 3 nodes)

Consecutive Lookups?



In Cache

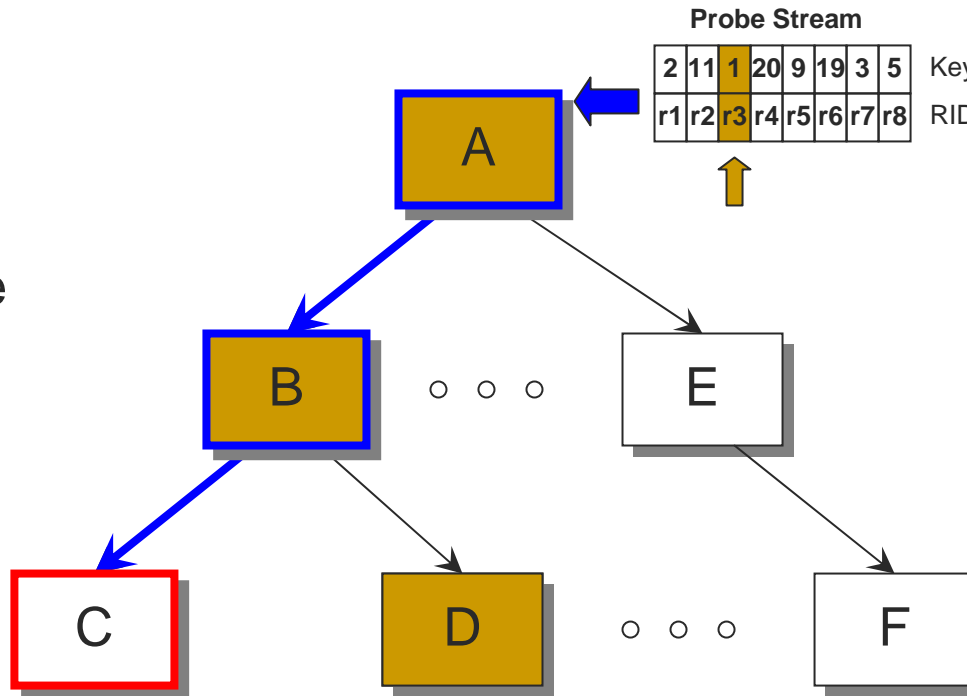


Cache Capacity
(Up to 3 nodes)

Consecutive Lookups?



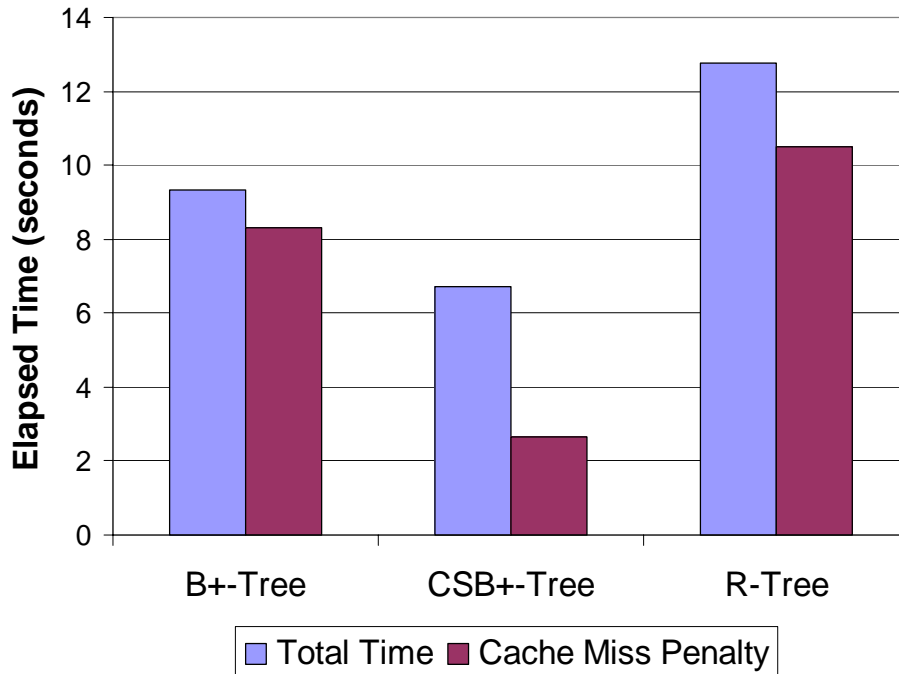
In Cache



Cache Capacity
(Up to 3 nodes)

Cache Thrashing!

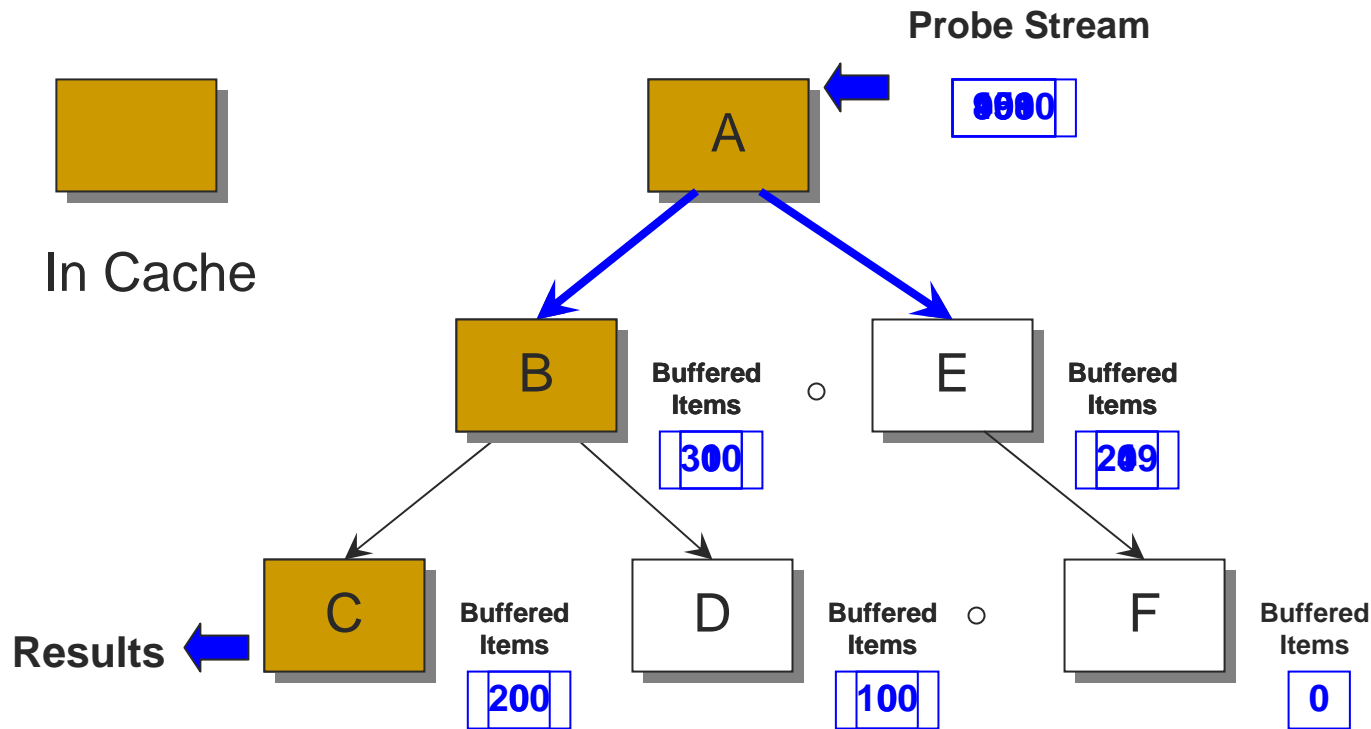
Index Cache Thrashing



	B+-tree	CSB+-tree	R-tree
Node Size	4KB	128B	4KB
Indexed items (inner)	5M keys		1M keys
Query entries (outer)	5M random searches		1M random searches

Both conventional and cache-conscious indexes suffer from index node thrashing!

Buffering Accesses to Index



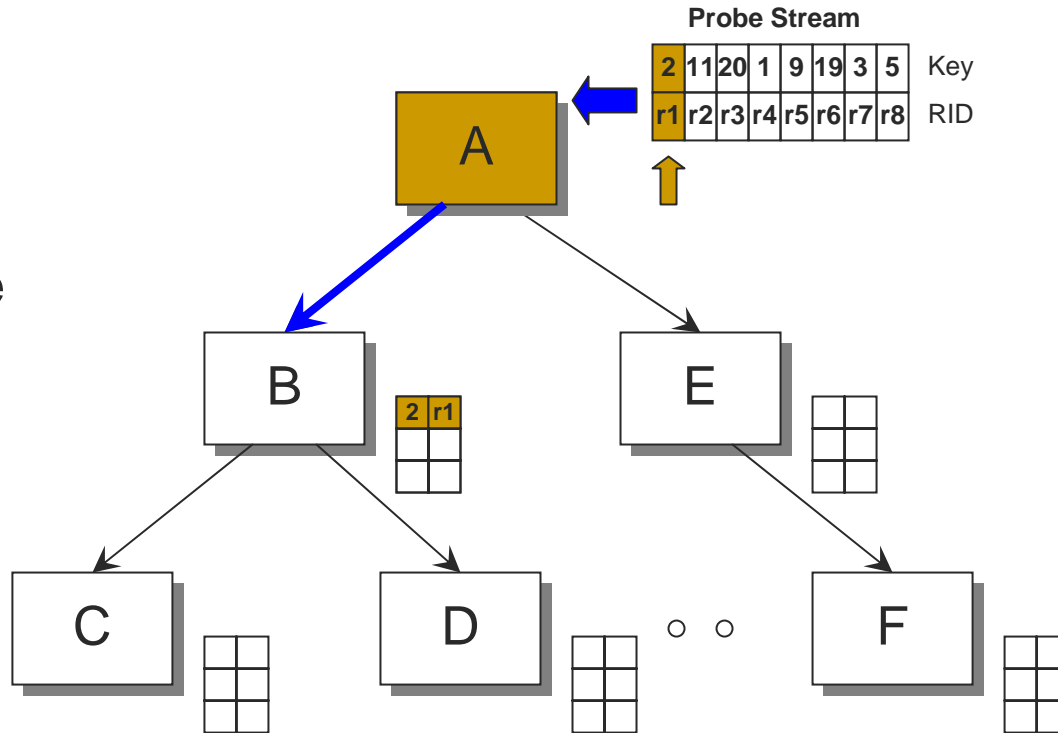
What is the point of buffering?

- ✓ Increases temporal and spatial locality
- ☹ Overhead: extra copy of data into buffers

Fixed-Size Buffering



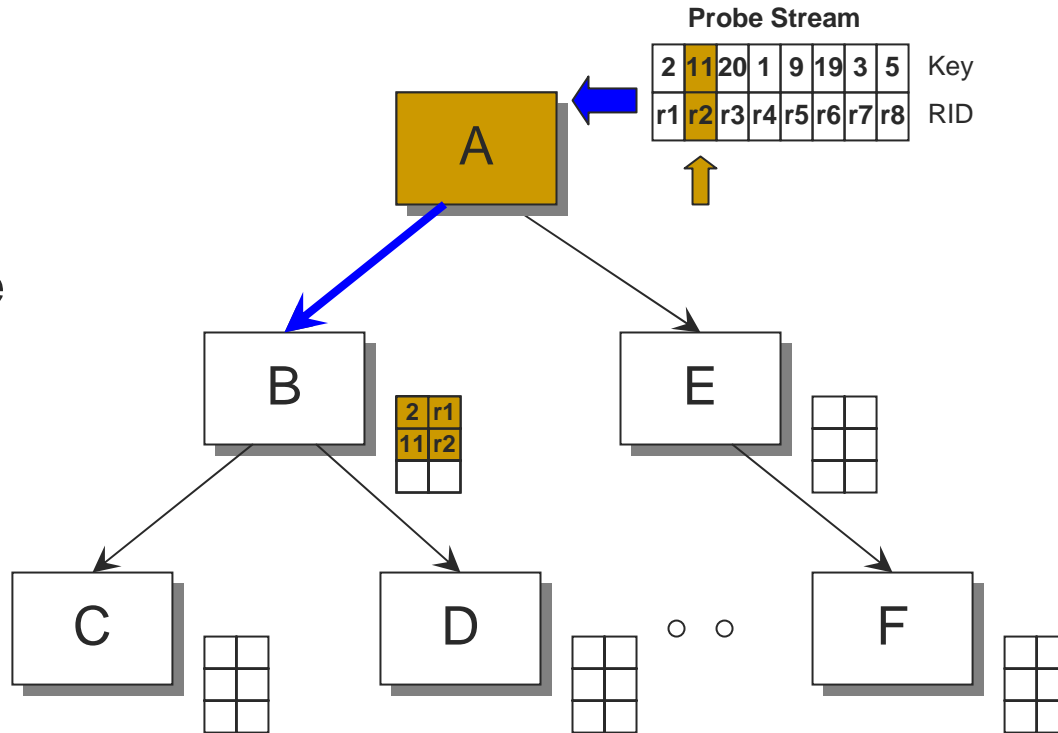
In Cache



Fixed-Size Buffering



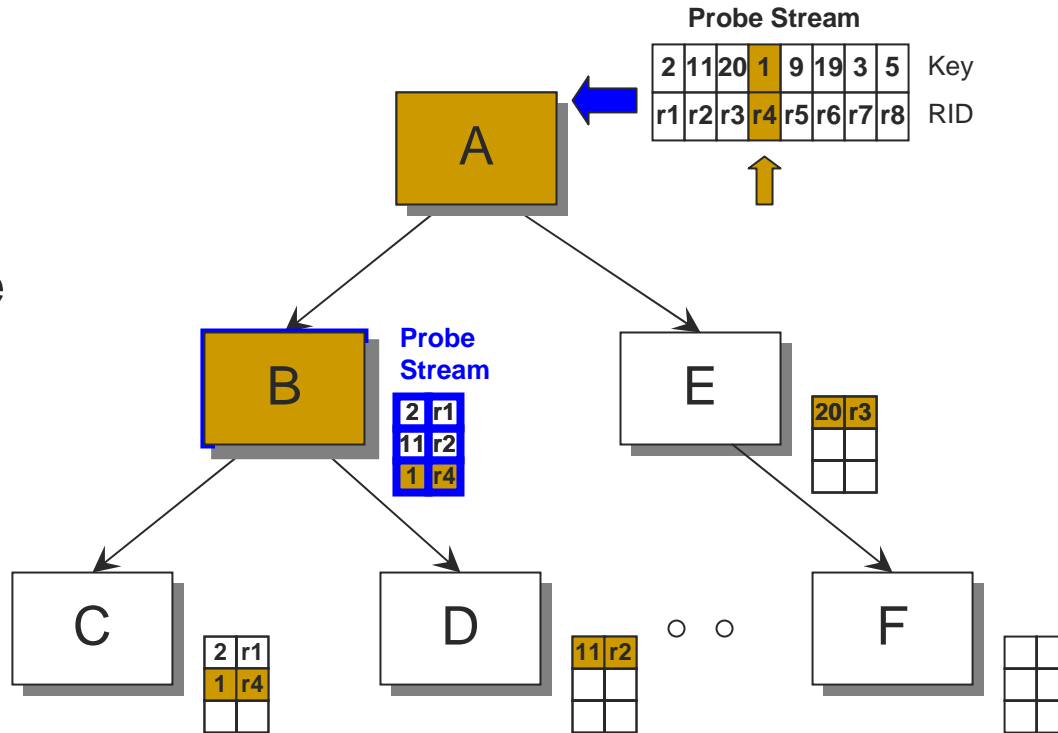
In Cache



Fixed-Size Buffering

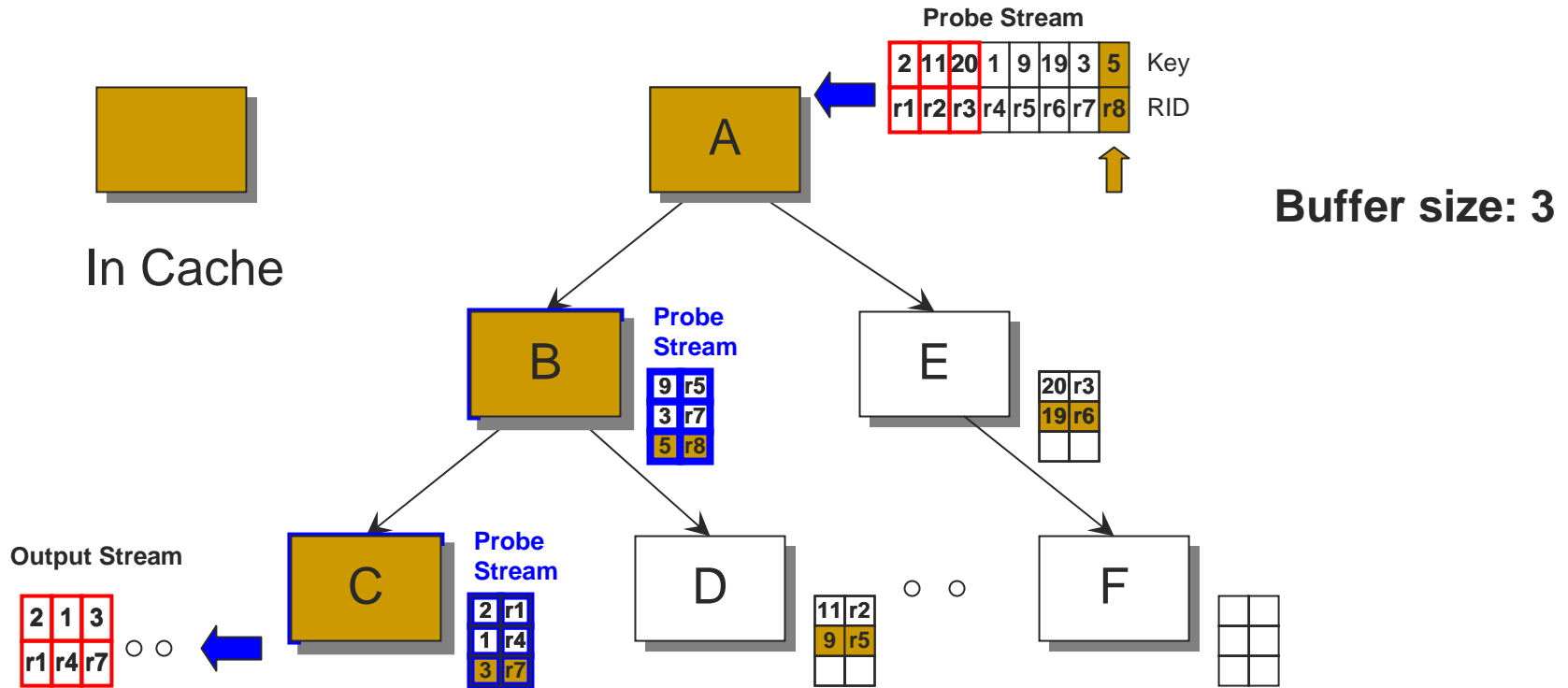


In Cache



Buffer size: 3

Fixed-Size Buffering

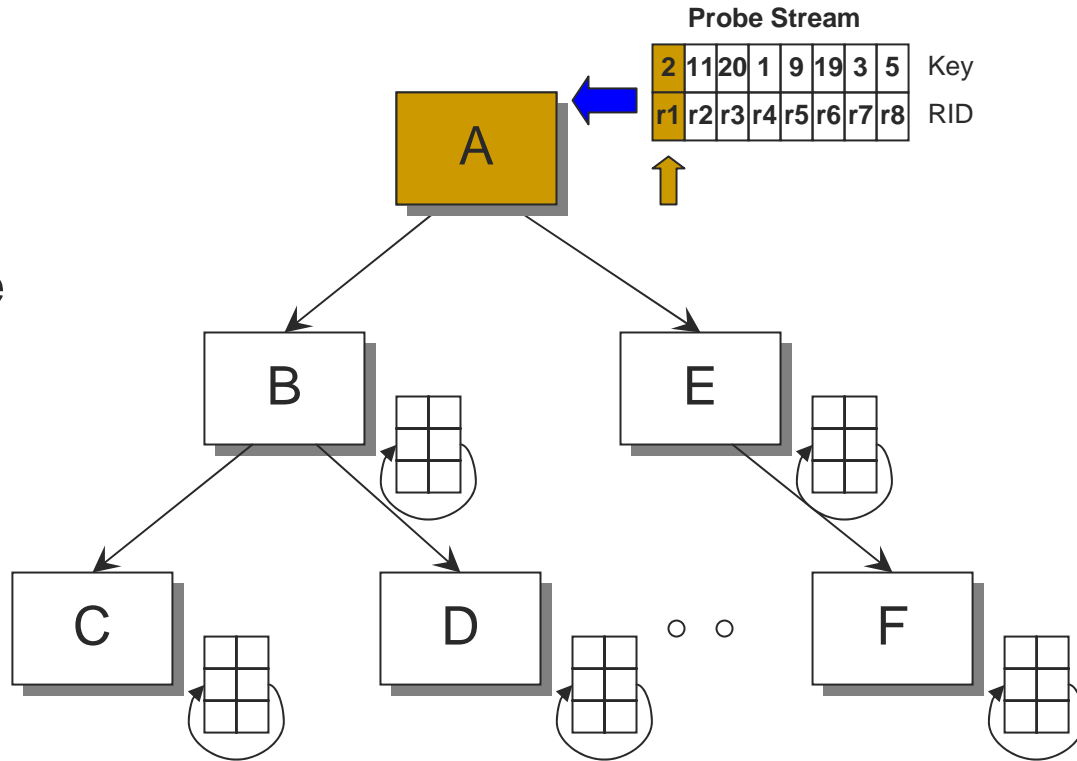


- *Eliminates a large number of cache misses, but not all of them*
- *Good control over the memory usage*

Variable-Size Buffering



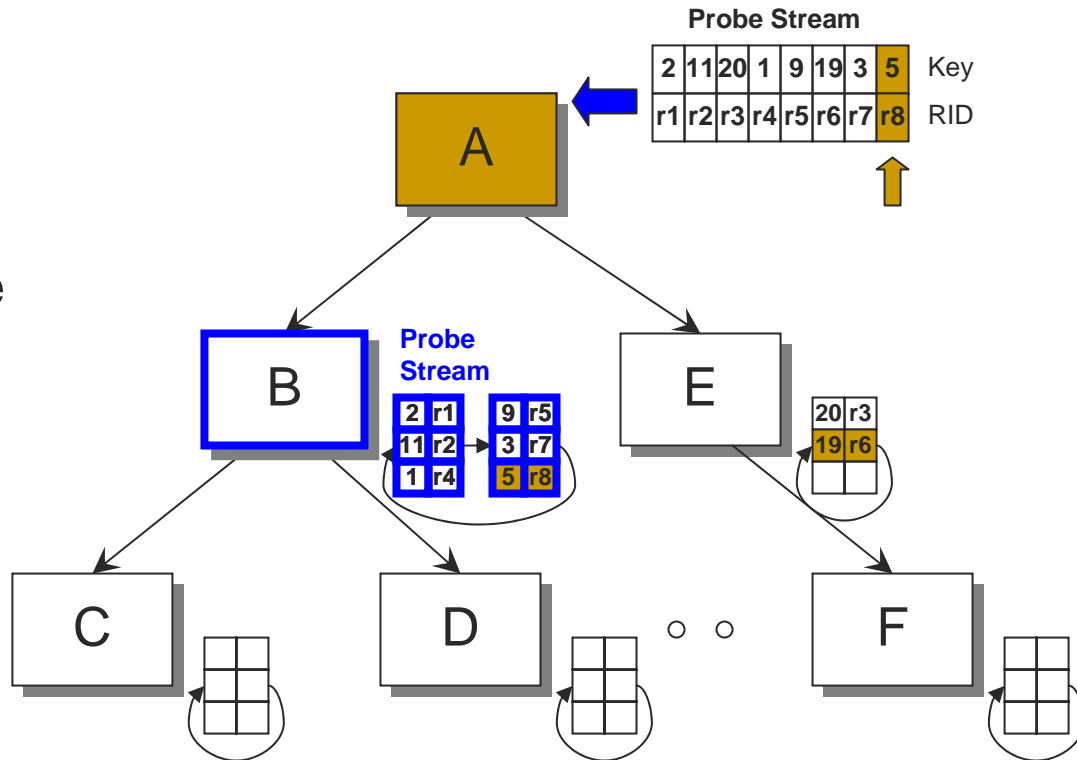
In Cache



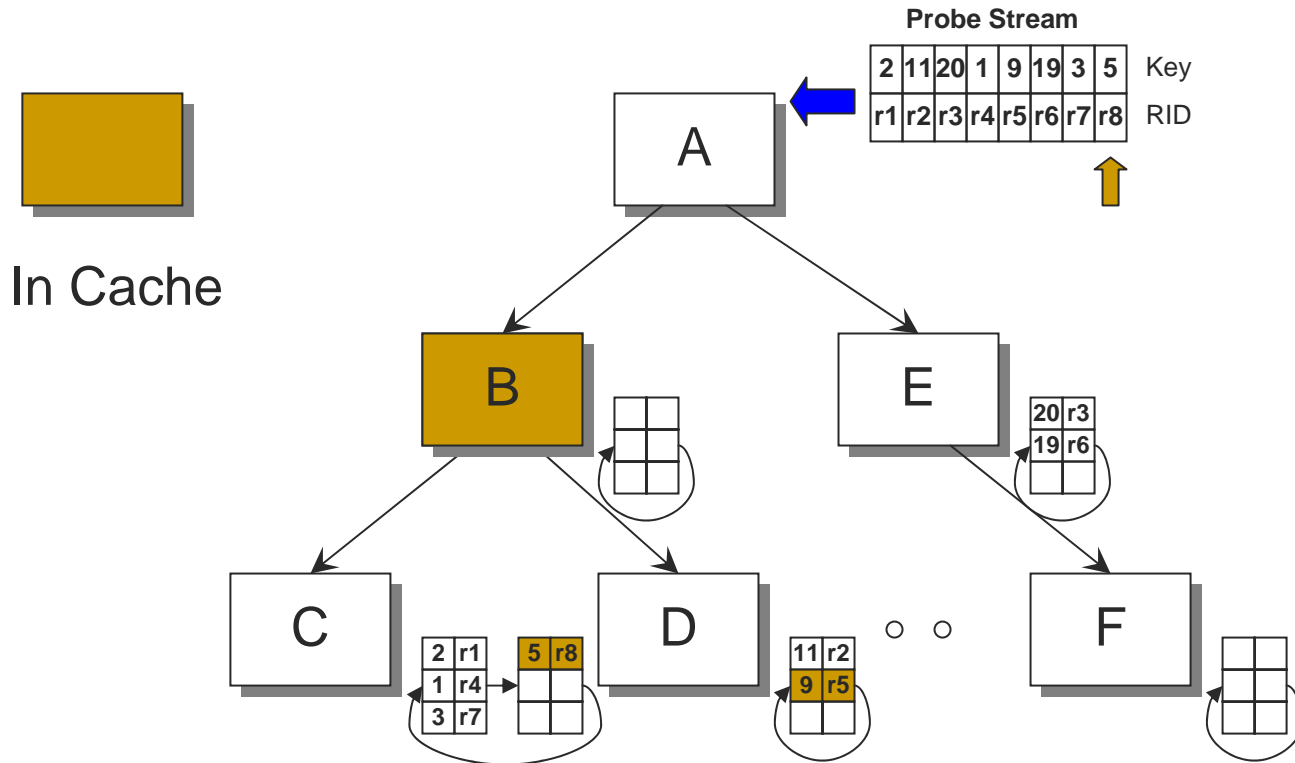
Variable-Size Buffering



In Cache

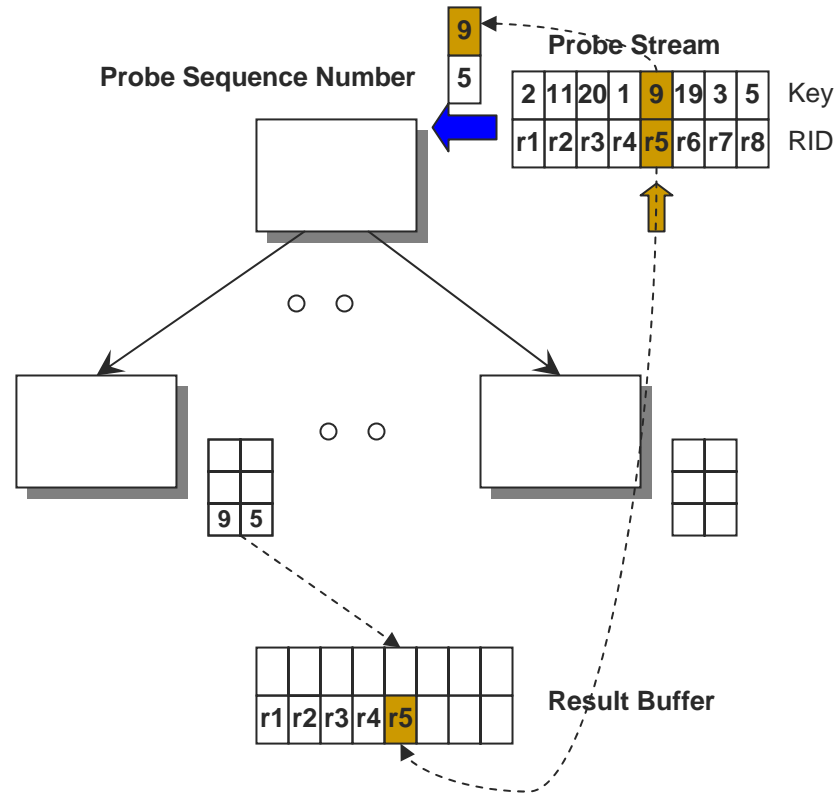


Variable-Size Buffering



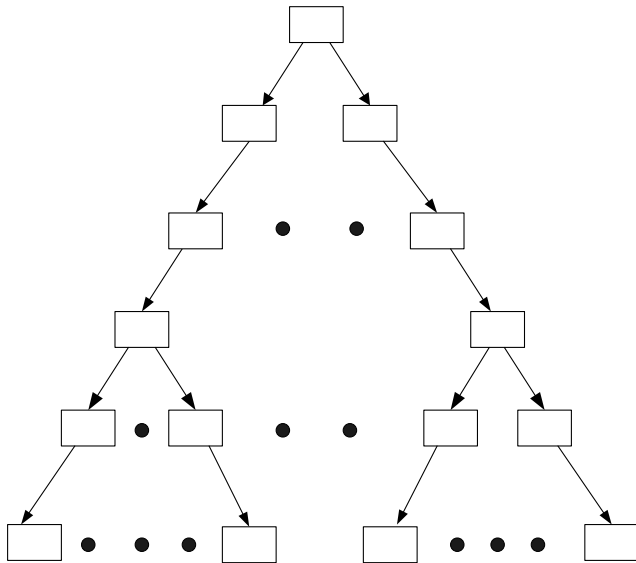
- *Completely eliminates index node cache thrashing by loading each node exactly once*
- *Memory requirement proportional to the size of input*

Order-Preserving Buffering

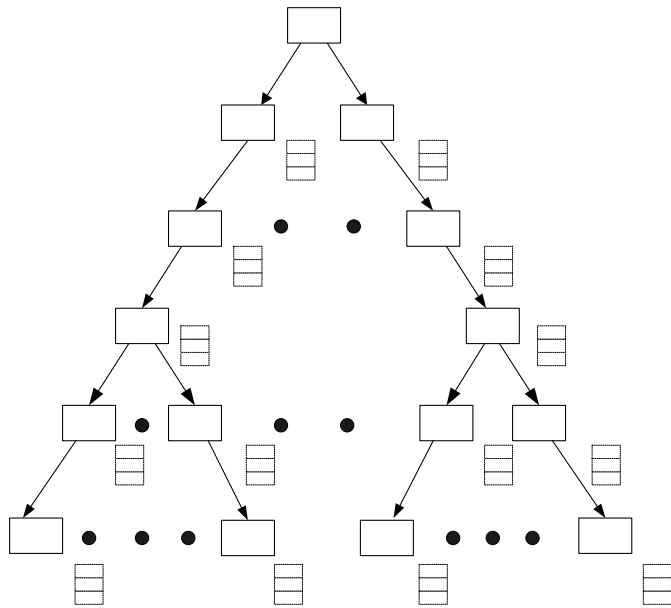


- *The output order matches the probe order*
- *The overhead to maintain the order is small*

[Where to Buffer?]

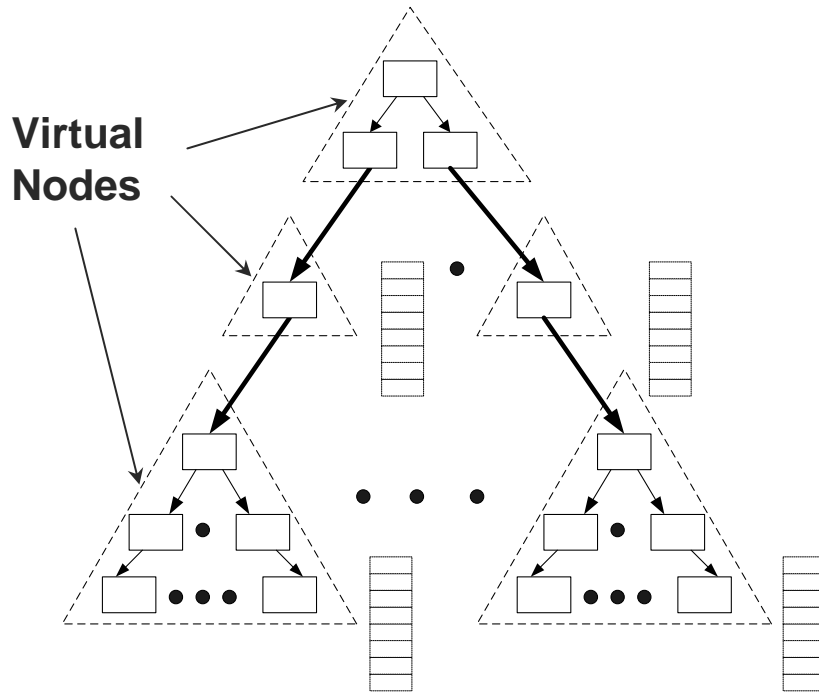


[Where to Buffer?]



Too much buffering!

Where to Buffer?



(2, 1, 3)

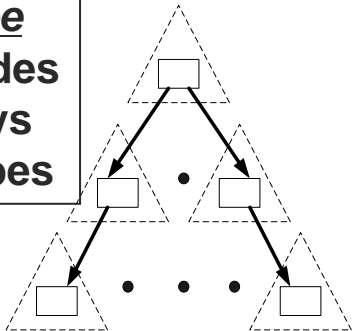
- By grouping multiple levels of a subtree into a virtual node, we reduce the amount of buffering
- Virtual nodes are units of buffering

Virtual Nodes Based on Cache Size

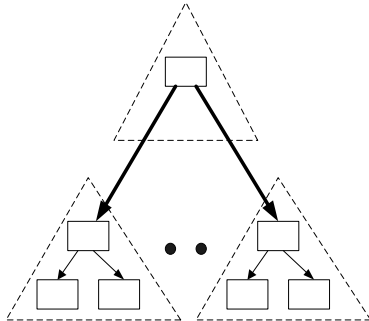
- Key idea: **biggest virtual nodes that fit in the cache**
 - Only count the effective size of a virtual node
 - Allow some cache memory for buffers
 - For internal virtual nodes, one tail cache line for each buffer should be counted
 - For leaf virtual nodes, results are not buffered
- Other factors:
 - TLB, Hardware prefetch, etc.

Buffer Placement (B⁺-Tree)

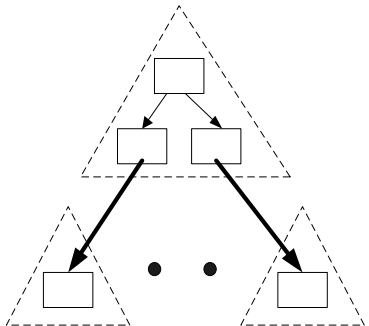
B⁺-Tree
 4KB Nodes
 5M Keys
 5M Probes



(1, 1, 1)

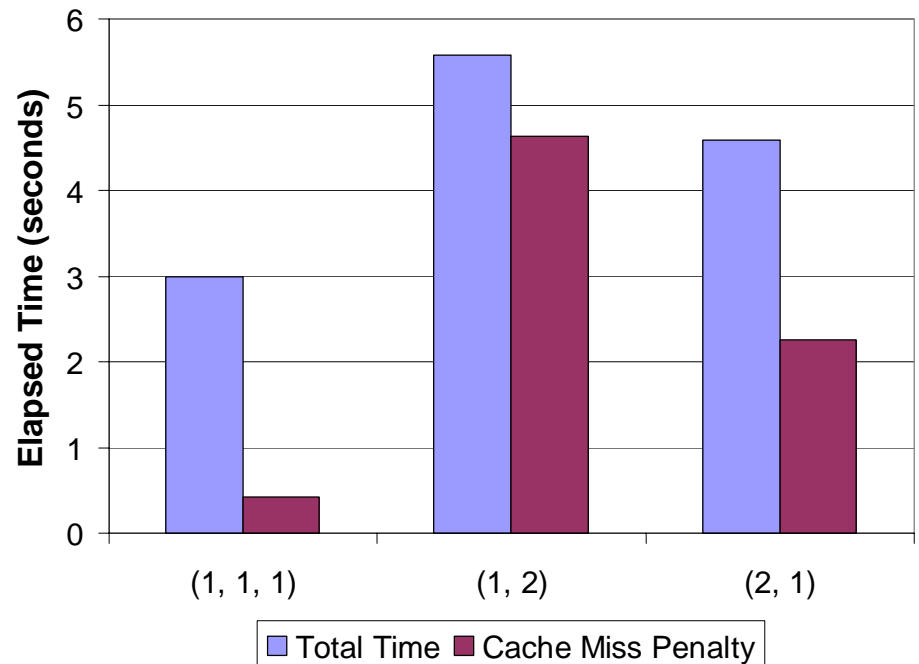


(1, 2)

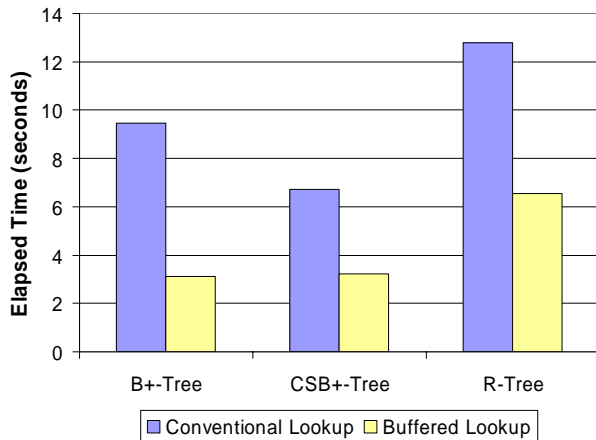


(2, 1)

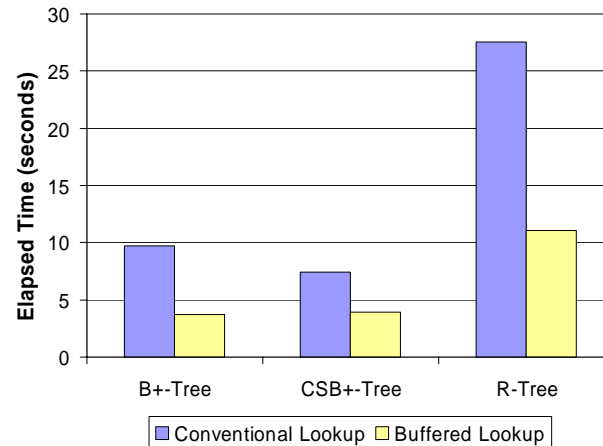
- L2 cache is too small to hold a two-level virtual node
- Cache analysis recommends (1, 1, 1)



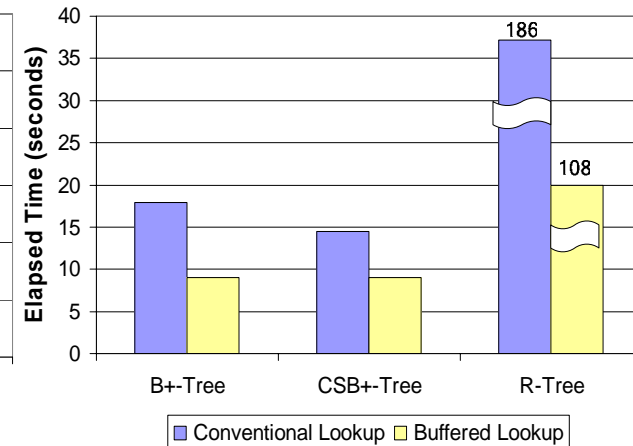
Overall Search Performance



Pentium 4



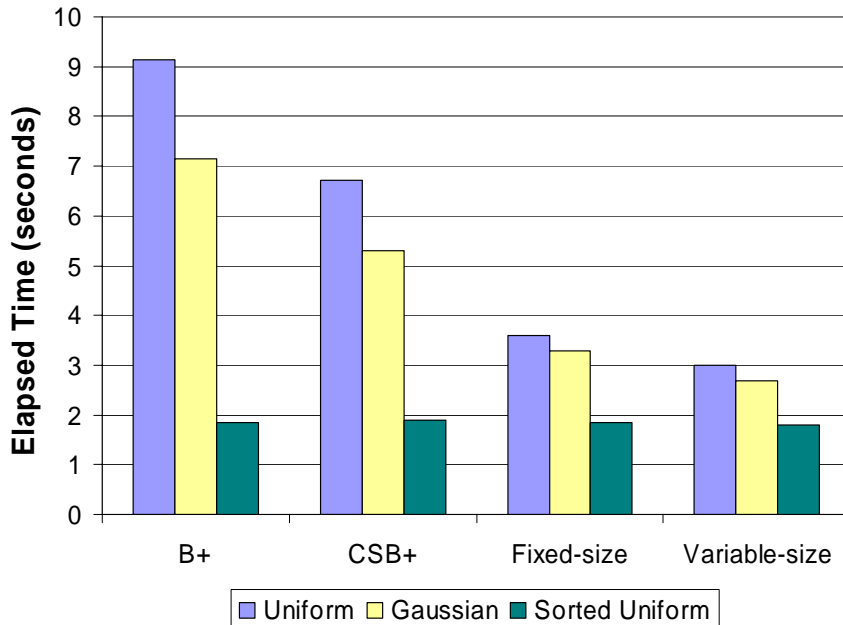
Pentium 3



Sun UltraSparc II

- A speedup of 2 to 3
- With buffering, B+-trees can achieve similar cache performance to CSB+-trees!
 - The buffering overhead is the same
 - The size of nodes is relatively unimportant for bulk access

Non-Random Probes

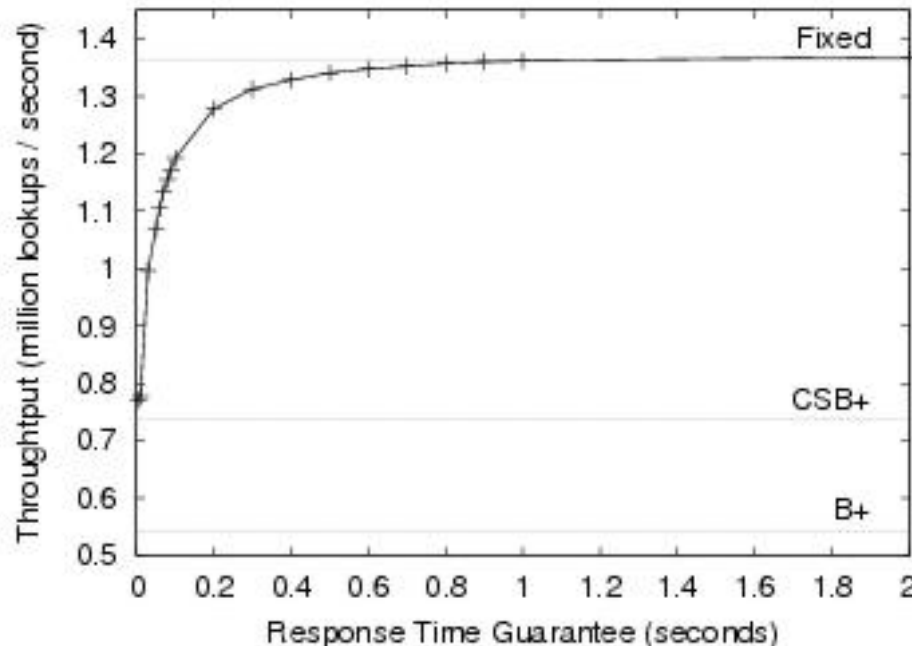


- *Uniformly distributed probes*
- *Gaussian distributed probes (95% probes touch 20% leaf nodes)*
- *Uniformly distributed probes sorted by the probe keys*

- When probes are skewed, the speedup is smaller
- If probes are sorted, all methods show good spatial locality
- The overhead of buffering is small

[Throughput vs. Response Time]

- Buffering does not guarantee fast response time
- Force the tree to be flushed after time interval t (called “response time guarantee”)



*Much Better
Throughput!*

[Summary (Part I)]

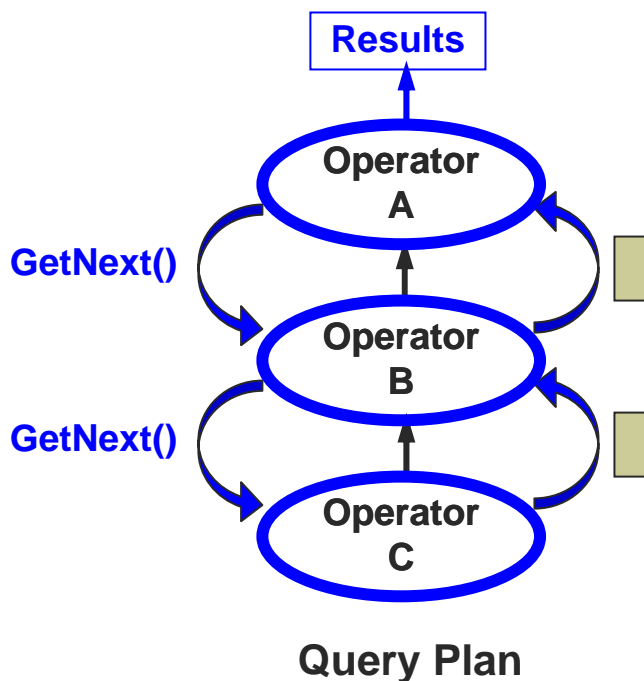
- Index structures suffer from cache thrashing between query accesses
- Buffering exploits cache spatial and temporal locality
 - For bulk lookups, our algorithms improve performance by a factor of two to three
- Relatively easy to implement
 - Buffering structures are separated from the index structures
 - Can be used for a variety of tree-based indexes

[Outline]

1. Improve data cache performance
 - Buffering memory accesses to index structures
2. **Improve instruction cache performance**
 - **Buffering intermediate results of a query subexpression**

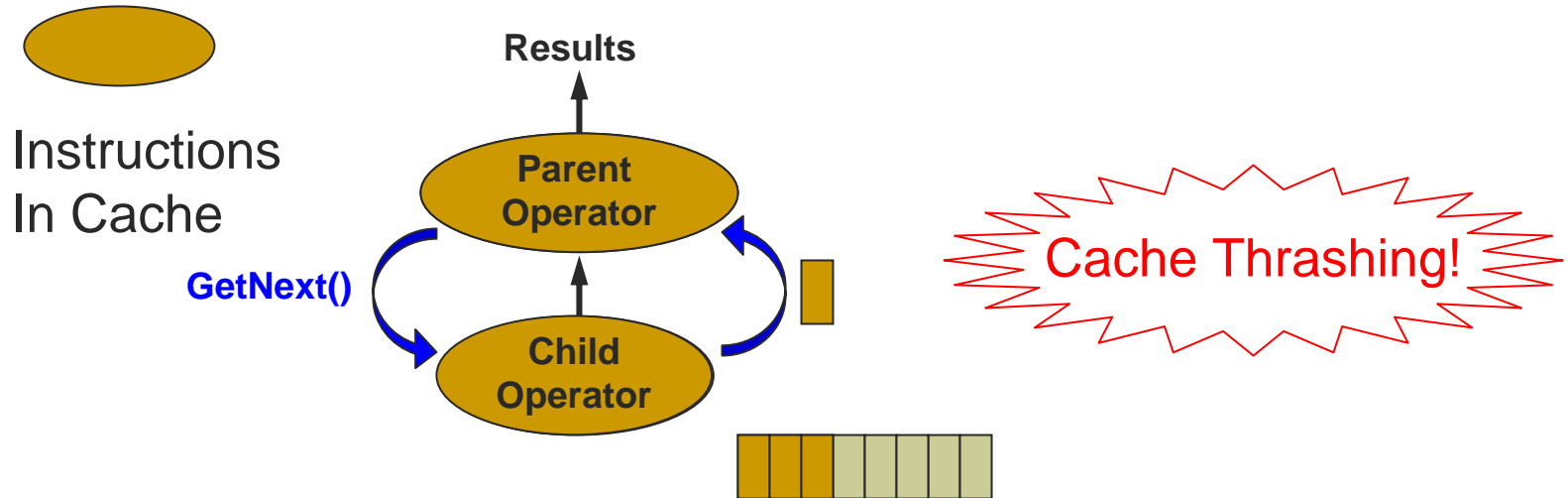
Demand-Driven Pipelined Execution Model

- Support **open-next-close** iterator interface
- An operator returns control immediately after generating one tuple



- Execution of operators are interleaved
- Advantages:
 - Can be executed by a single process or thread
 - Avoid inter-process communication
 - Avoid process synchronization and scheduling
 - Minimize data copies and keep only the current data items in memory

Instruction Cache Performance

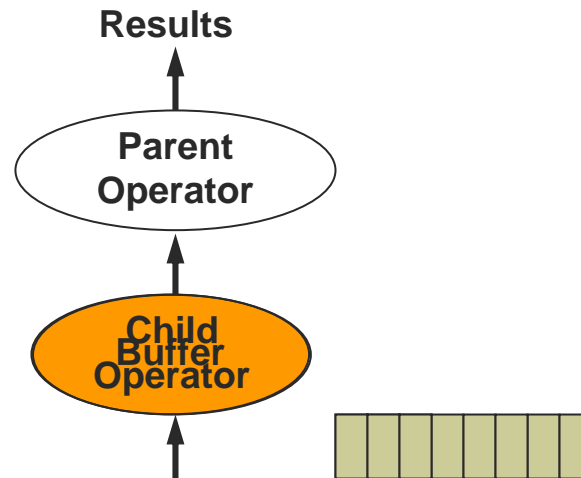


- Poor instruction locality
- Secondary issues:
 - Poor branch predication accuracy

[Buffering Operations]



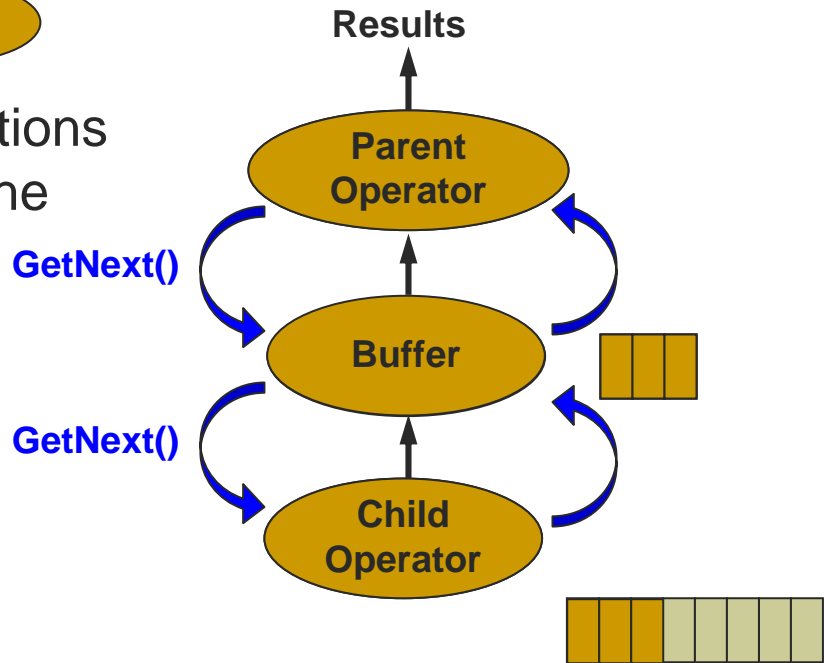
Instructions
In Cache



[Buffering Operations]



Instructions
In Cache

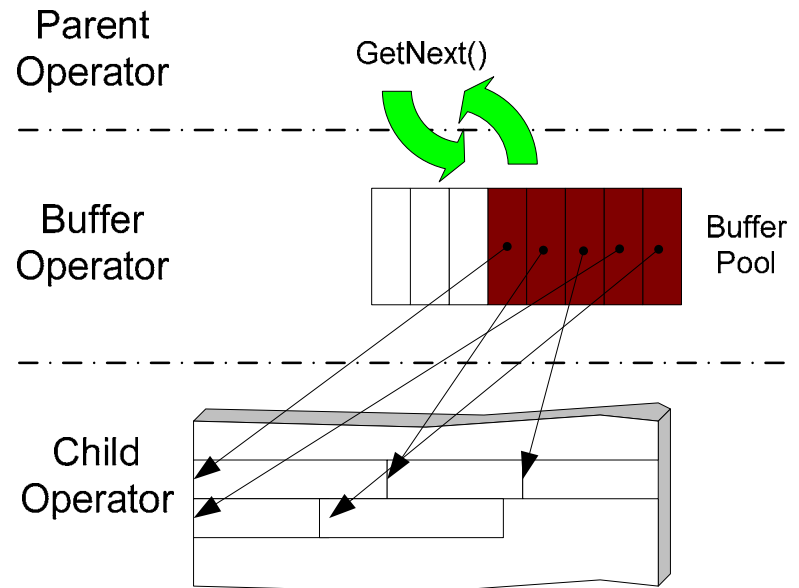


*Execution of operators
are no longer interleaved!*

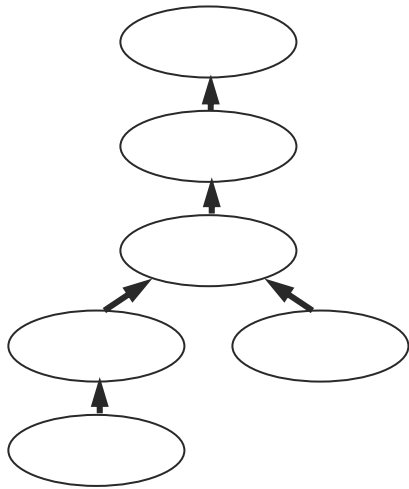
Improve instruction locality!

Buffer Operator

- Supports **open-next-close** interface
- Store pointers to intermediate tuples
- Intermediate tuples are stored in child operator's memory space
- ✓ No (or little) modifications to other operators
- ☹ The child operator needs more memory to store intermediate tuples
 - ☹ More L2 data cache misses

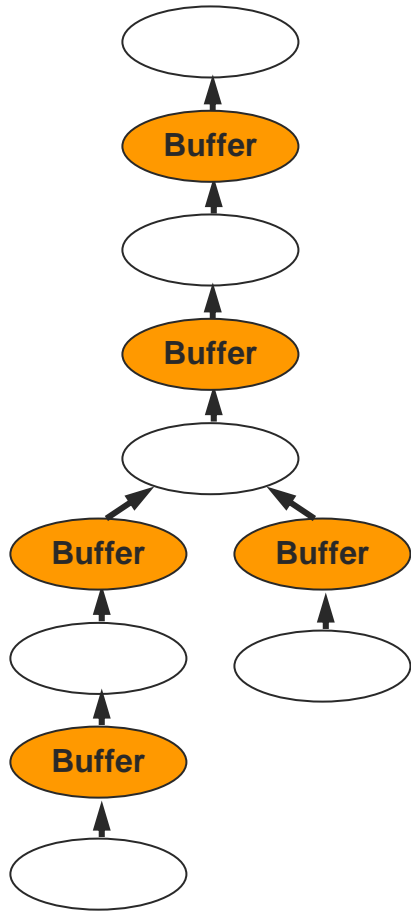


[Where to Buffer?]



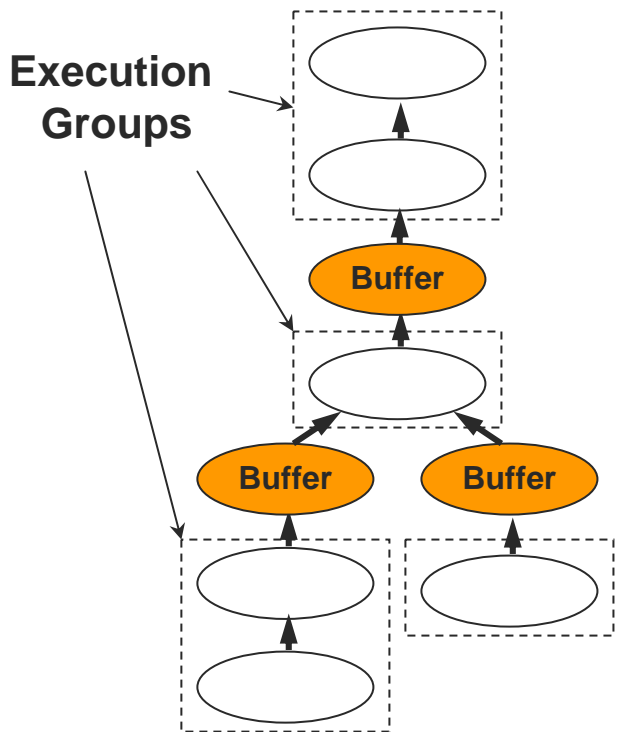
Original Query Plan

[Where to Buffer?]



Too much buffering!

Where to Buffer?



Buffered Query Plan

- By grouping operators into an execution group, we reduce the amount of buffering
- Execution groups are units of buffering

Execution Groups

Based on Instruction Footprints

- Key idea: **biggest execution groups that fit in the instruction cache**
 - Compute the combined instruction footprint of an execution group
 - “Execution Group” is similar to “Virtual Node” in principle
- Don't buffer for
 - Blocking operators (sorting, hash-table building, etc)
 - Already buffer query executions below them
 - Operators with small cardinalities

[Overall Algorithm]

- Accept a query plan from optimizer
- Consider only non-blocking operators with large output cardinalities
- A **bottom up** pass is made of the query plan to create execution groups
 - The combined instruction footprint of an execution group is less than L1 cache size
- Add a buffer operator above each execution group

Instruction Footprint Analysis

- Static call graphs?
- Dynamic call graphs
 - Execution path is usually data independent
 - Calibrate the core database system with a set of simple queries
- Postgres (memory-resident) footprint analysis
 - Different modules share functions
 - Pentium 4 has a 16K L1 instruction cache

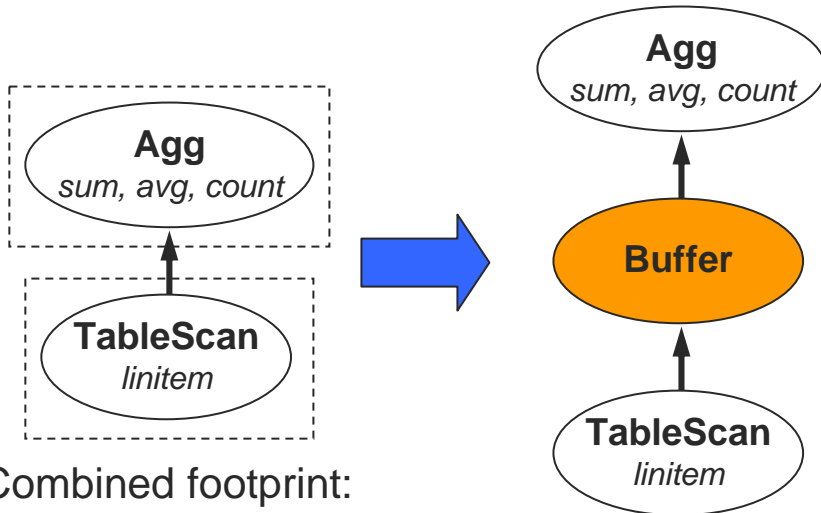
Table Scan		9K ~ 13K
Index Scan		14K
Sort		14K
NestLoop Join	TableScan	11K
	IndexScan	11K
Merge Join		12K
Hash Join	Build	10K
	Probe	12K
Agg	Base	10K
	COUNT	< 1K
	MAX, MIN	1.6K
	SUM	2.7K
	AVG	6.3K
Buffer		< 1K

[Validation: Query 1]

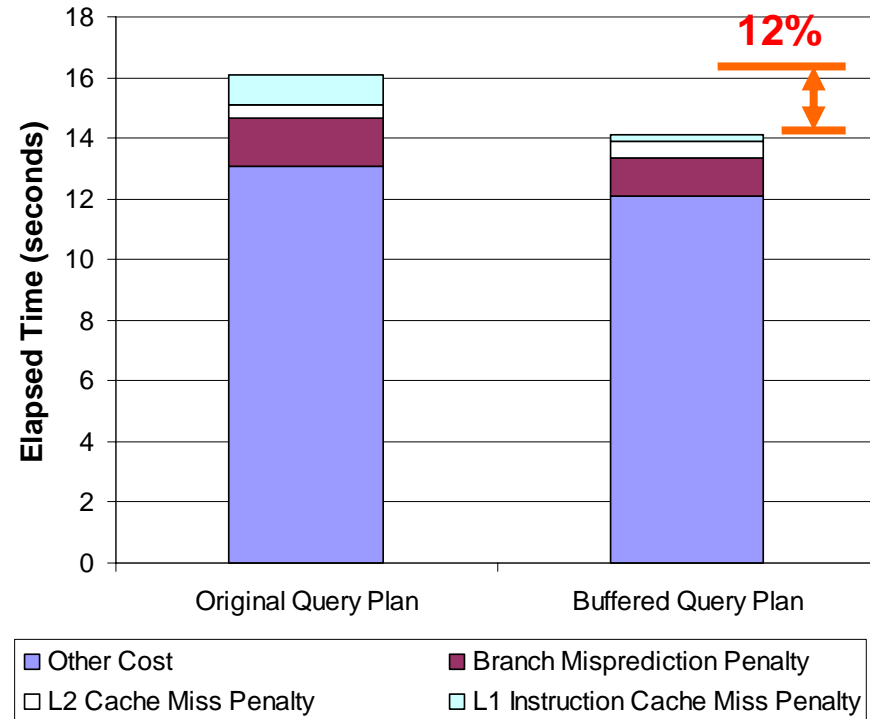
```

SELECT SUM(l_extendedprice *
          (1 - l_discount)
          (1 + l_tax)) AS sum_charge,
       AVG(l_quantity) AS avg_qty,
       COUNT(*) AS count_order
FROM lineitem
WHERE l_shipdate <= date '1998-11-01'
(TPC-H)

```



Combined footprint:
23K > 16K L1



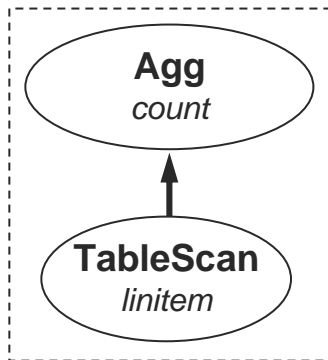
The buffered plan reduces

- L1 instruction cache miss by 80%

Validation: Query 2

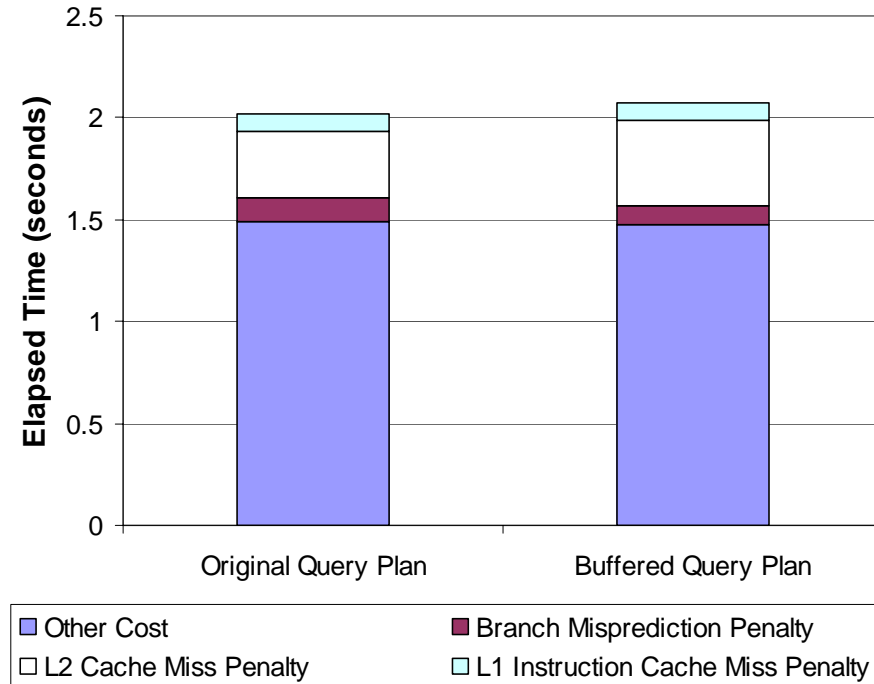
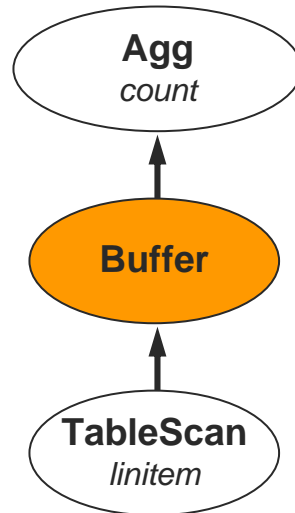
```
SELECT COUNT(*) AS count_order
FROM lineitem
WHERE l_shipdate <= date '1998-11-01'
(TPC-H)
```

Recommends
no buffering!



Combined footprint:
15K < 16K L1

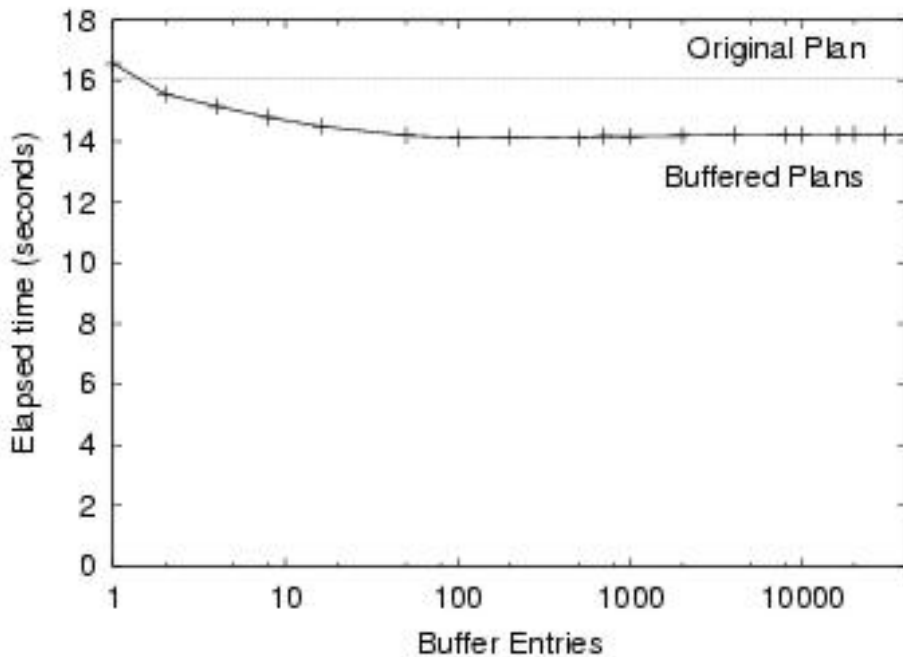
?



- No improvement
- Recommendation is right!
- Low overhead too

Buffer Size

- The benefit is roughly proportional to $1/\text{buffersize}$
- Bigger buffer sizes may incur more L2 cache misses



[A Join Example: Query 3]

```
SELECT SUM(o_totalprice) AS sum_price,  
       AVG(l_discount) AS avg_discount,  
       COUNT(*) AS count_order
```

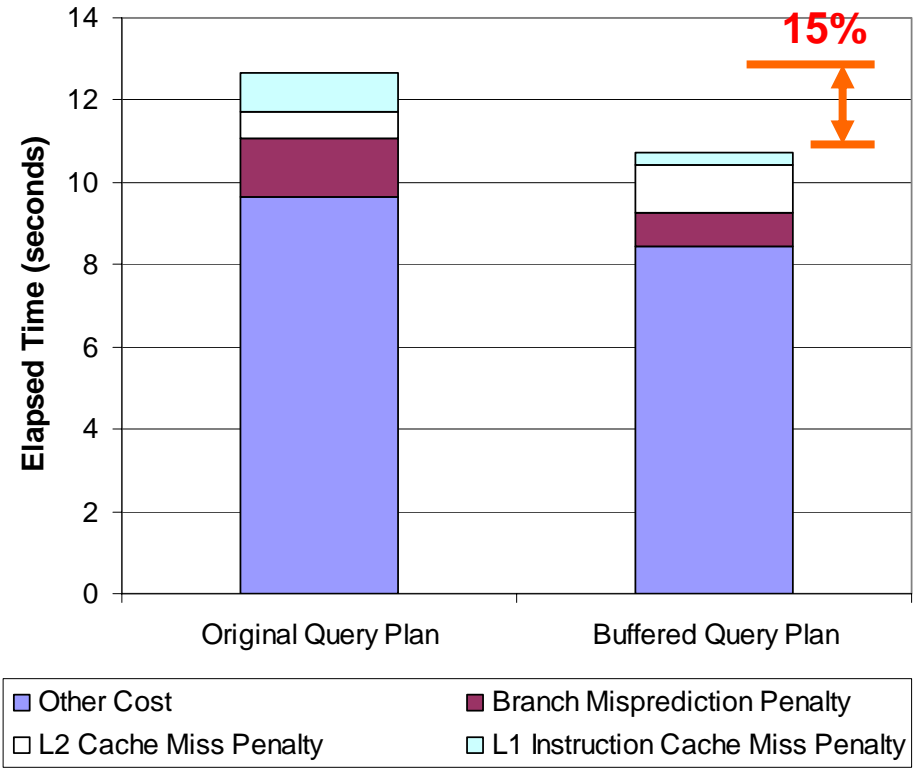
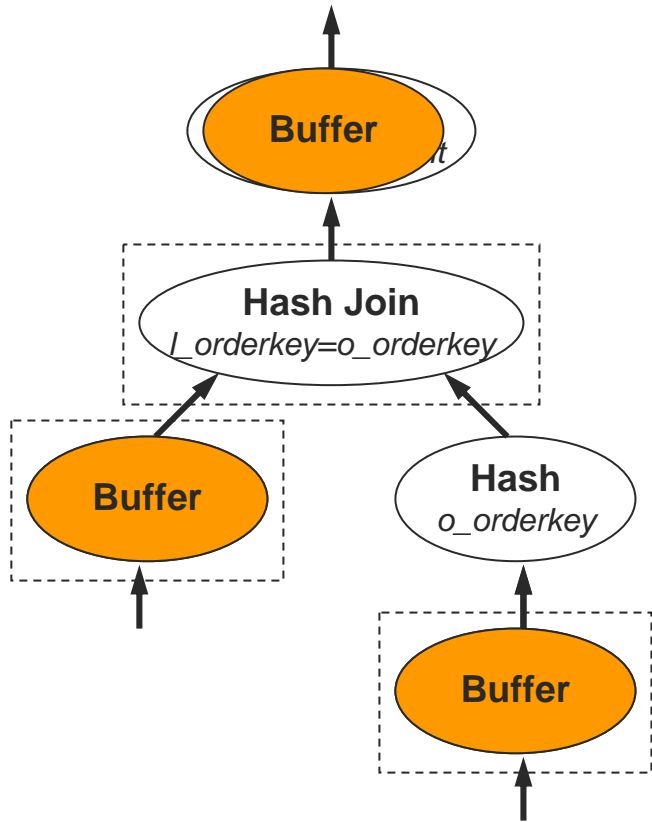
```
FROM lineitem, orders
```

```
WHERE l_orderkey = o_orderkey
```

```
AND    l_shipdate <= date '1998-11-01'
```

```
(TPC-H)
```

[Hash Join]



The buffered plan reduces

- L1 instruction cache miss by 70%

Summary (Part II)

- The conventional query execution model may suffer from instruction cache thrashing
- Buffering exploits instruction cache spatial and temporal locality
 - Especially useful for complex queries with large instruction footprints
- Relatively easy to implement
 - No significant change of other operators
 - The decision of based on instruction footprints

Cache Optimization Via Buffering

Cache	<u>Data cache</u>	<u>Instruction cache</u>
Objective	Tree-based index structures	Query execution
Buffers	Index probes	Intermediate results
Buffer unit	Virtual nodes	Execution groups
Based on	Index node effective size	Instruction footprint
Improves	D-Cache locality	I-Cache locality

Buffering improves cache performance of DBMS!

[Future Work]

- Consider buffering within a query optimizer
 - Design a detailed cost model
- For data cache performance
 - Hash indexes: buffering probes using a prefix of the hashed key
- For instruction cache performance
 - Break a large operator into several sub-operators? (e.g. complex aggregation)
 - Reschedule execution?

[Thank you!]

- For more information
 - www.cs.columbia.edu/~kar