HOT Inside

The Technical Architecture

Pavan Deolasee

May 22, 2008

Overview

- PostgreSQL MVCC
- Motivation for Improvement
- HOT Basics
- HOT Internals
- Limitations
- Performance Numbers and Charts

What Does HOT Stand For ?

- Heap Organized Tuples
- Heap Optimized Tuples
- Heap Overflow Tuples
- Heap Only Tuples

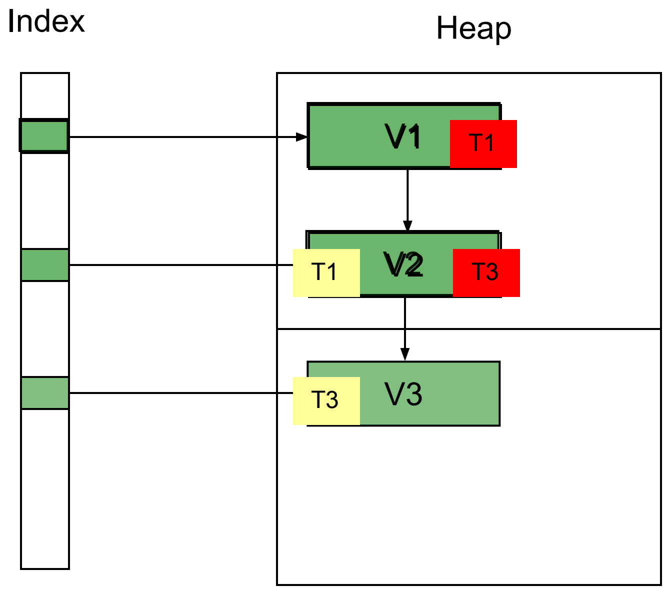
Credits

- Its not entirely my work
- Several people contributed, some directly, many indirectly
  - Simon Riggs – for writing initial design doc and getting me involved
  - Heikki – for code review, idea generation/validation and participating in several long discussions.
  - Tom Lane – for patch review and code rework
  - Korry – for extensive code review within EnterpriseDB
  - Dharmendra, Siva, Merlin – for testing correctness/performance
  - Florian, Gregory – for floating ideas
  - Denis, Bruce – for constant encouragement and making me rework HOT thrice ☺
  - Faiz, Hope – for excellent project management within EnterpriseDB
  - Nikhil – for hearing to all my stupid ideas and helping with initial work
- The list is so long that I must have missed few names – apologies and many thanks to them

Some Background - MVCC

- PostgreSQL uses MVCC (Multi Version Concurrency Control) for transaction semantics
- The good things:
  - Readers don't wait for writers
  - Writer doesn't wait for readers
  - Highly concurrent access and no locking overhead
- The bad things:
  - Multiple versions of a row are created
  - The older, dead versions can not be easily removed because indexes don't have visibility information
  - Maintenance overhead to reduce table/index bloat

MVCC - UPDATE

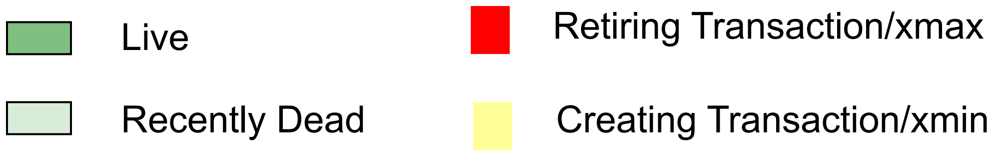


- Transaction T1 Updates V1
- Transaction T1 Commits

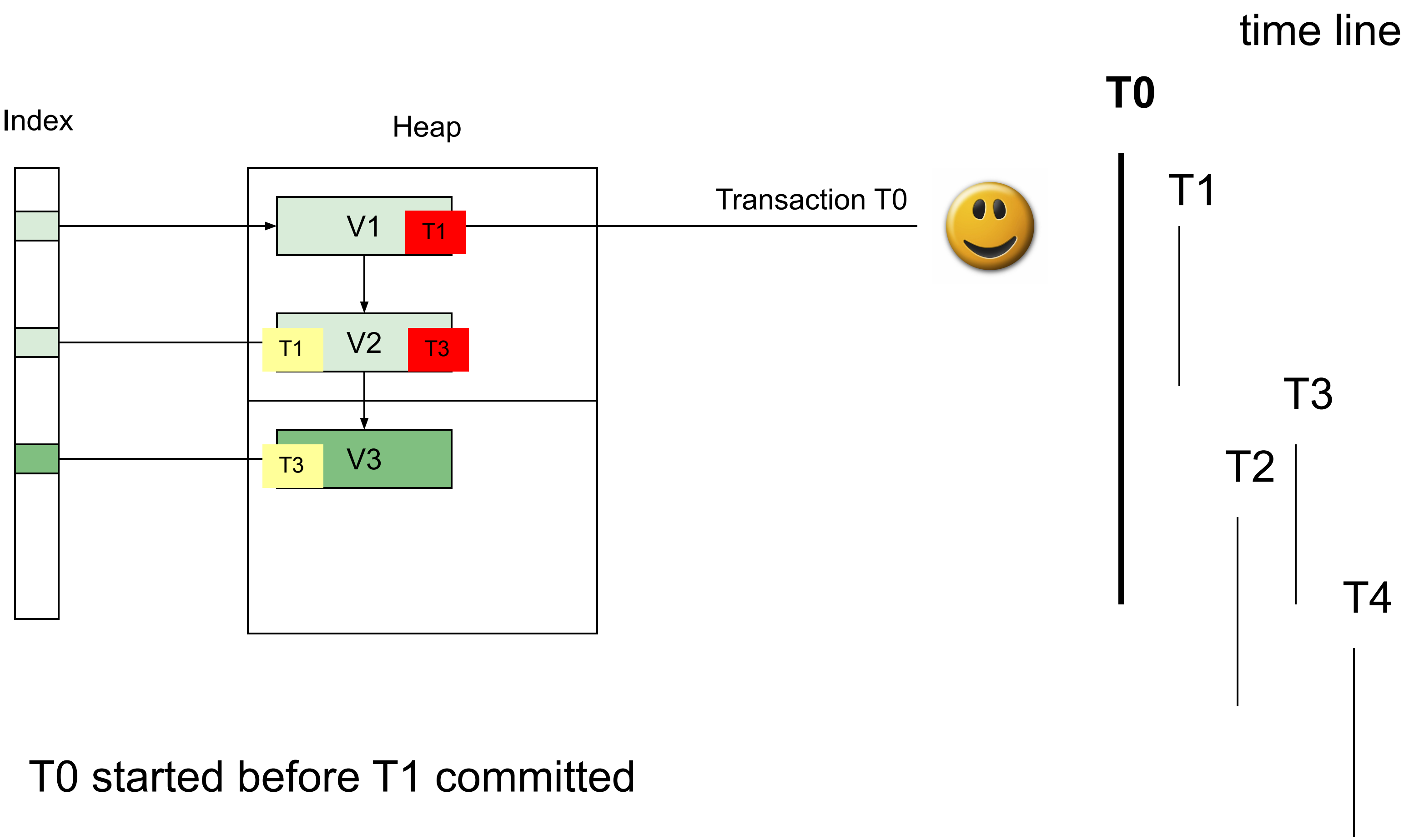
  V1 is dead, but still visible to older transactions, so we call it **RECENTLY DEAD**

- Transaction T3 Updates V2
- Transaction T3 Commits

  V2 is dead, but still visible to older transactions, It's also **RECENTLY DEAD**



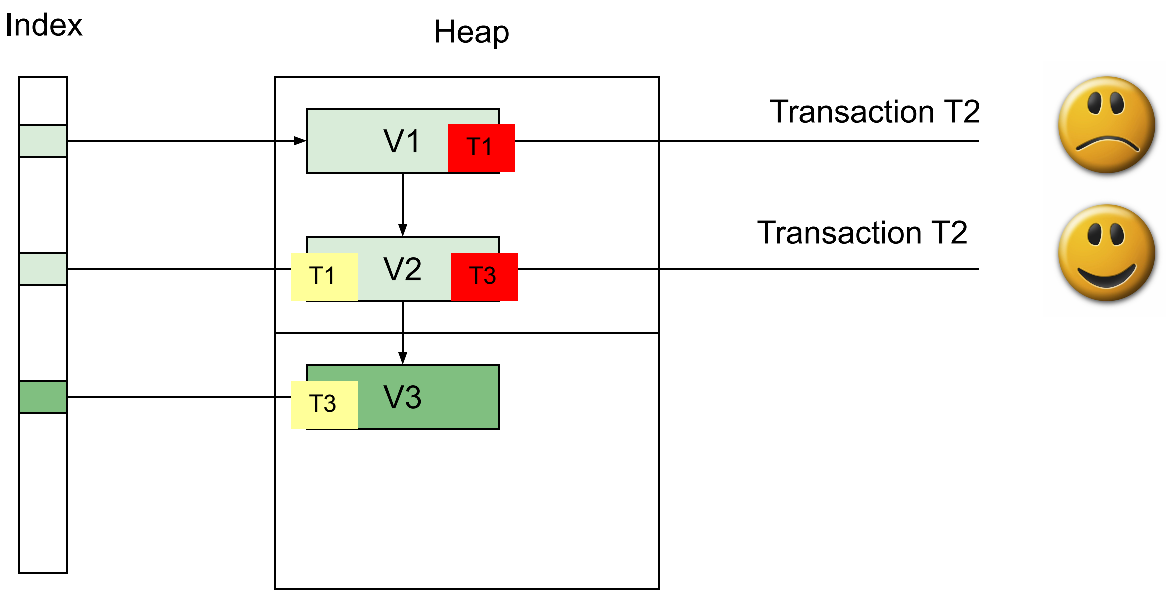
MVCC - Visibility

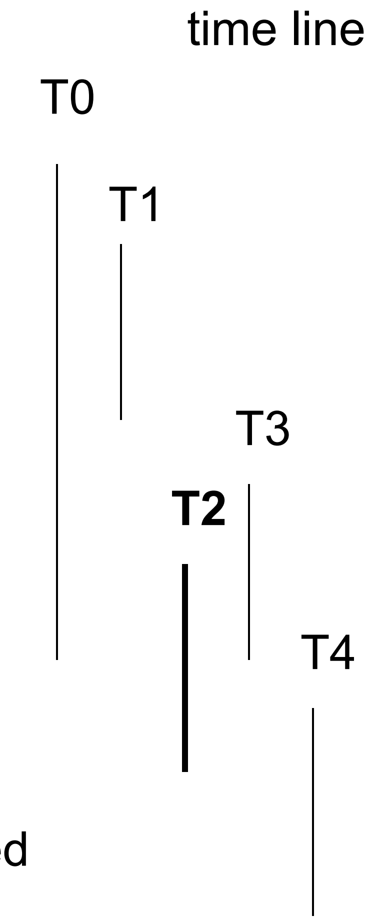


T0 started before T1 committed

T0 can only see V1

MVCC - Visibility

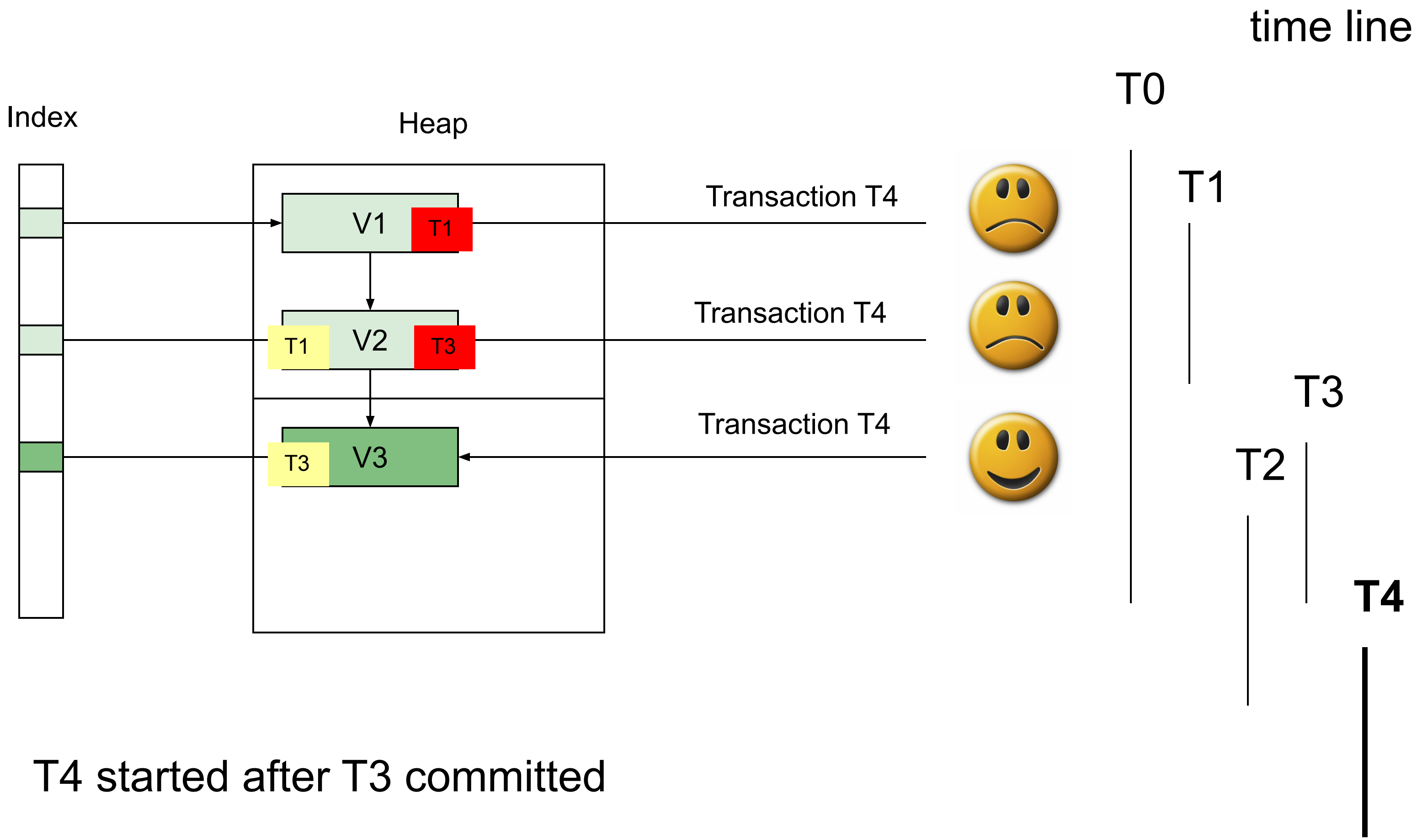




T2 started after T1 committed, but before T3 committed

T2 can only see V2

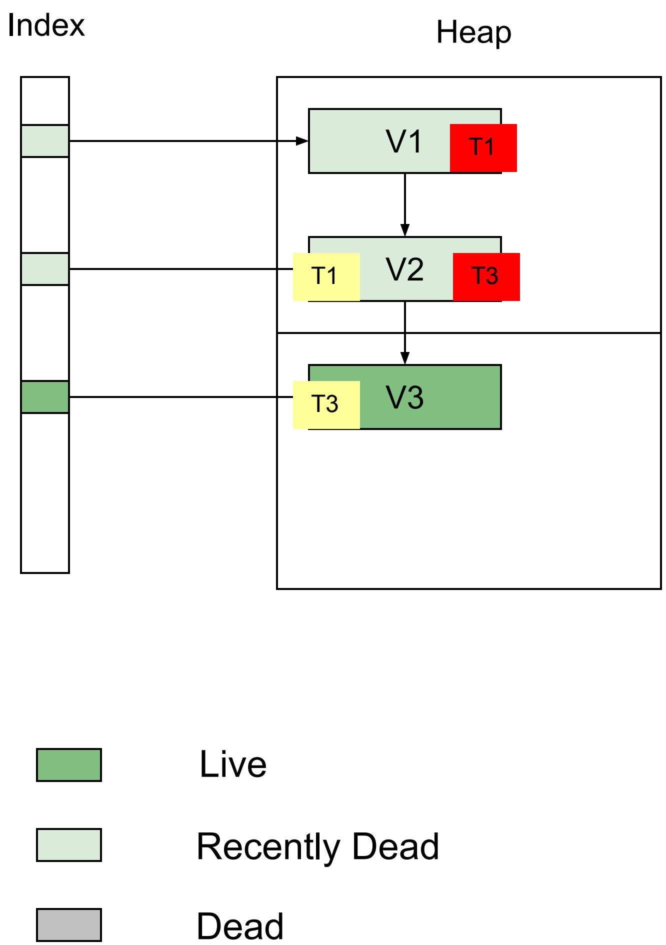
MVCC - Visibility



T4 started after T3 committed

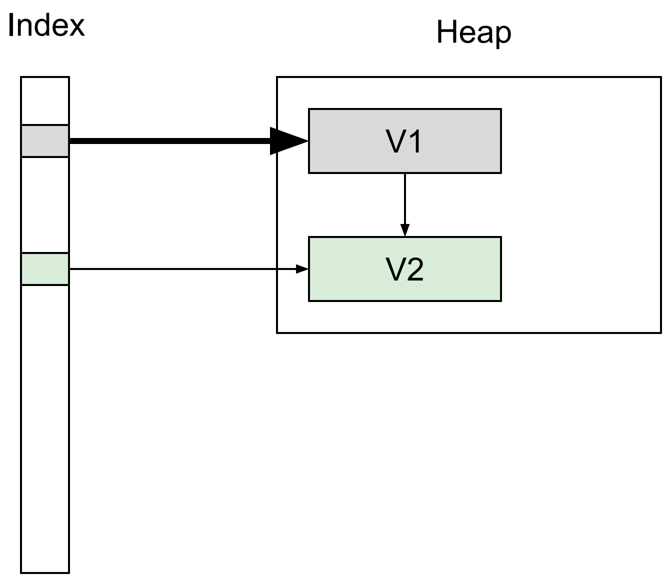
T4 can only see V3

MVCC – Tuple States



- V1 and V2 are **RECENTLY DEAD**, V3 is the most current and **LIVE** version
- V1 and V2 can not be removed, because T0 and T2 can still see them
- T0 finishes, V1 becomes **DEAD**
- T2 finishes, V2 becomes **DEAD**
- Only V3 remains **LIVE**

Removing DEAD Tuples

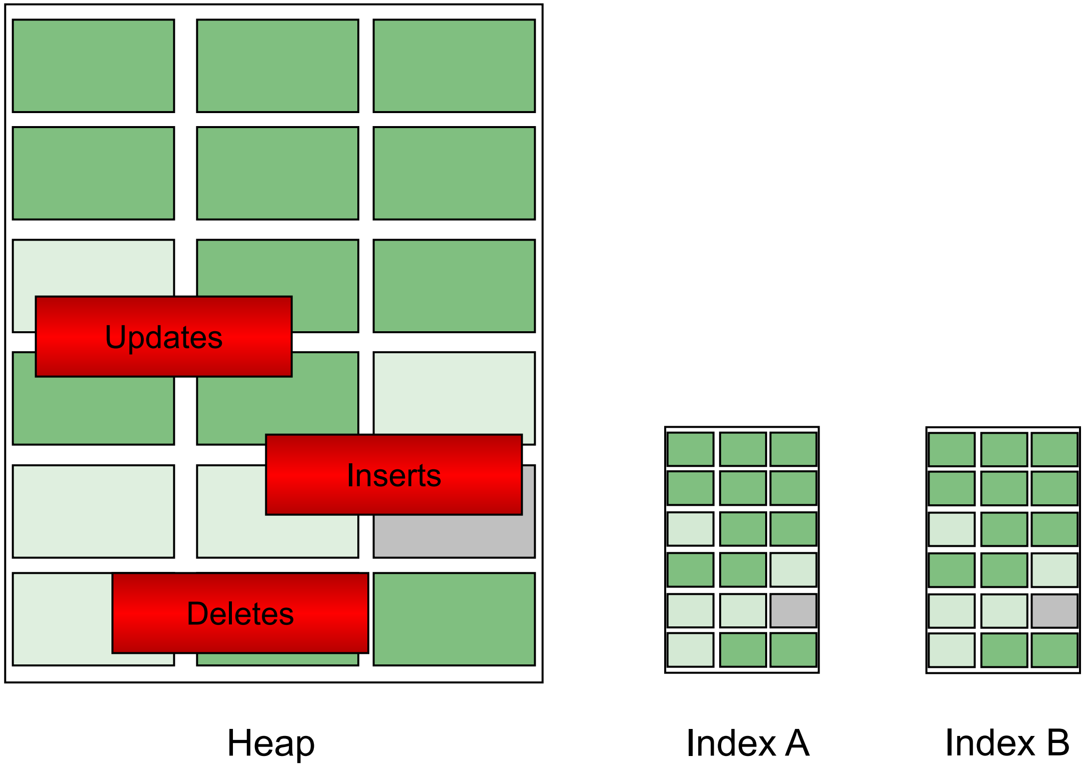


- V1 is DEAD. If it's removed, we would have a dangling pointer from the index.
- V1 can not be removed unless the index pointers pointing to it are also removed

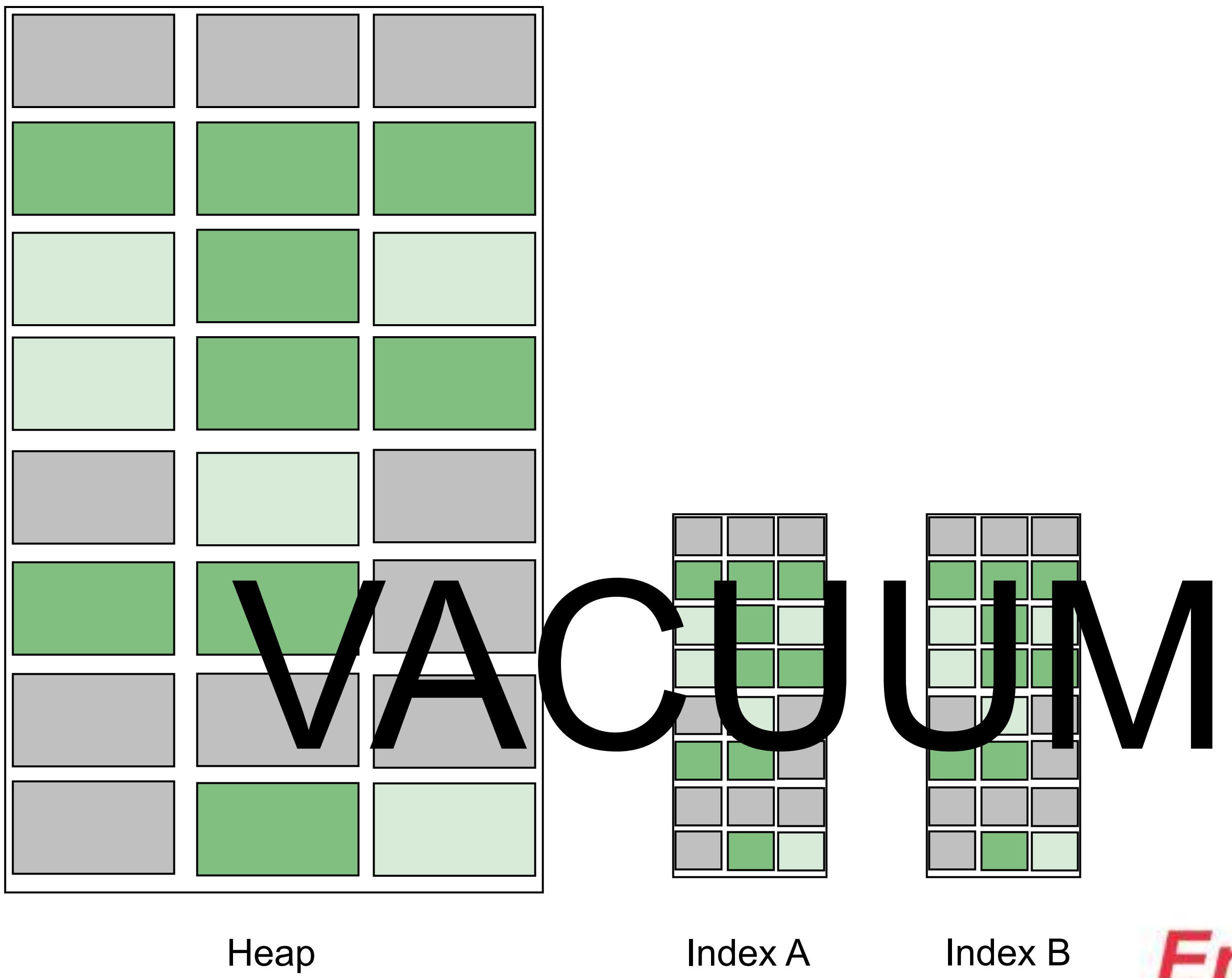
Note: Index entries do not have any visibility Information

- Near impossible to reliably find index pointers of a given tuple.

MVCC - Index/Heap Bloat

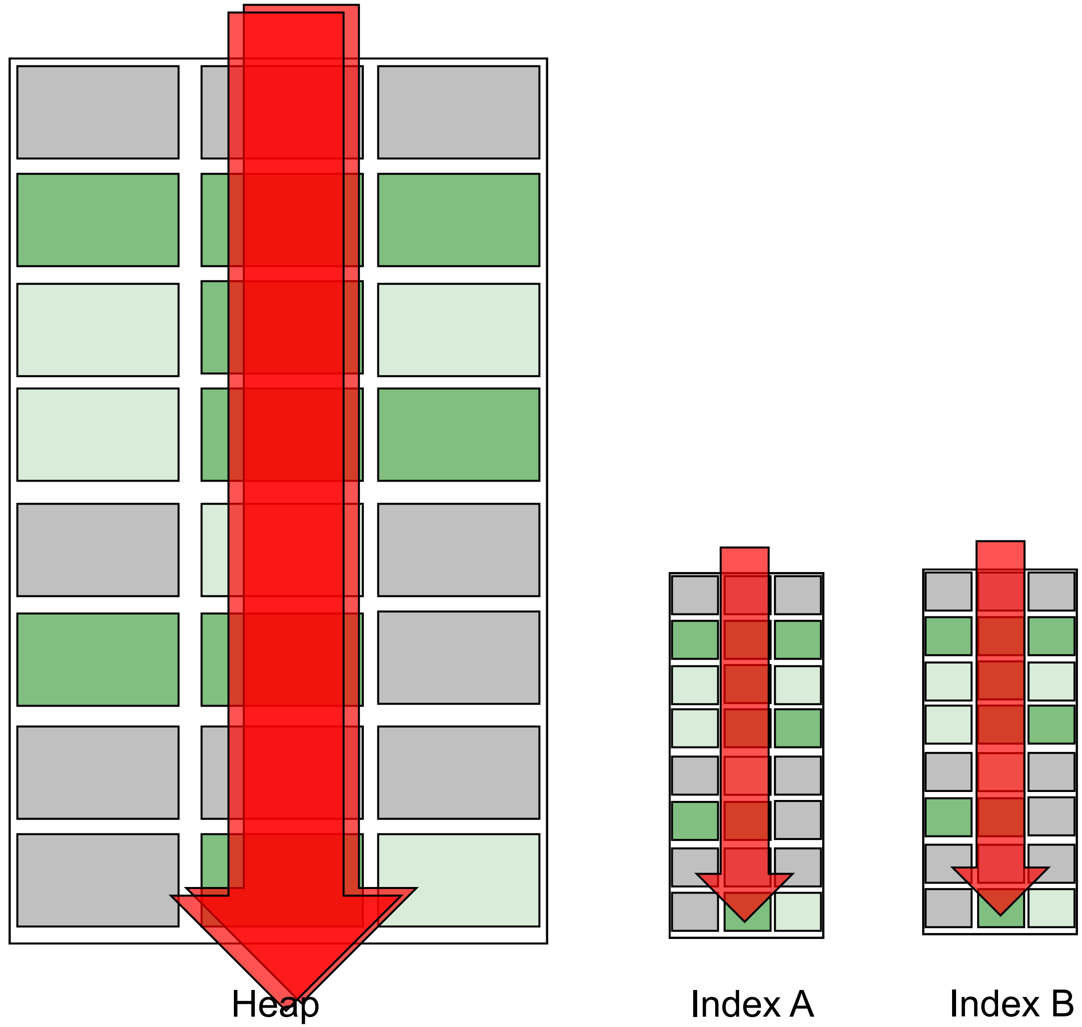


MVCC - Index/Heap Bloat



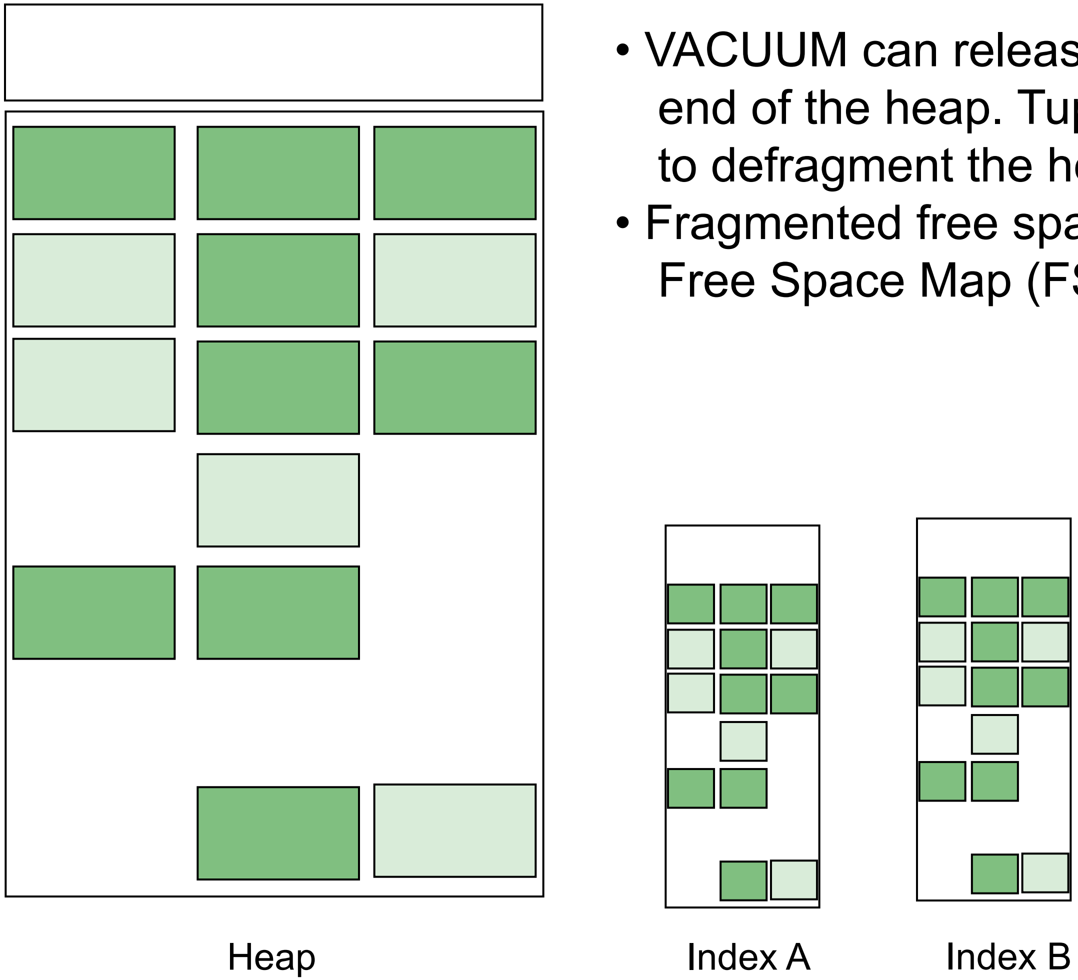
EnterpriseDB™

Vacuum – Two Phase Process



Vacuum

- VACUUM can release free space only at the end of the heap. Tuples are not reorganized to defragment the heap
- Fragmented free space is recorded in the Free Space Map (FSM)



EnterpriseDB™

Motivation

- Frequent Updates and Deletes bloat the heap and indexes resulting in performance degradation in long term – spiral of death
- Each version of a row has it's own index entry, irrespective of whether index columns changed or not – index bloat
- Retail VACUUM is near impossible (dangling index pointers)
- Regular maintenance is required to keep heap/index bloat in check (VACUUM and VACUUM FULL)
  - Normal VACUUM may not shrink the heap, VACUUM FULL can but requires exclusive lock on the table
  - VACUUM requires two passes over the heap and one or more passes over each index.
  - VACUUM generates lots of IO activity and can impact the normal performance of the database.
  - Must be configured properly

Pgbench Results

- scale = 90, clients = 30, transactions/client = 1,000,000
- two CPU, dual core, 2 GB machine
- separate disks for data (3 disks RAID0) and WAL (1 disk)
- shared_buffers = 1536MB
- autovacuum = on
- autovacuum_naptime = 60
- autovacuum_vacuum_threshold = 500
- autovacuum_vacuum_scale_factor = 0.1
- autovacuum_vacuum_cost_delay = 10ms
- autovacuum_vacuum_cost_limit = -1

Heap Bloat (# blocks)

	Postgres 8.2		Postgres 8.3 Pre HOT		Postgres 8.3 Post HOT	
	Original Size	**Increase in Size**	Original Size	Increase in Size	Original Size	Increase in Size
Branches	1	**49,425**	1	**166**	1	**142**
Tellers	6	**18,080**	5	**1,021**	5	**171**
Accounts	155,173	**243,193**	147,541	**245,835**	147,541	**5,523**

In 8.2, the heap bloat is too much for small and large tables

EnterpriseDB™

Postgres 8.3 – Multiple Autovacuum

	Postgres 8.2		**Postgres 8.3 Pre HOT**		**Postgres 8.3 Post HOT**	
	Original Size	**Increase in Size**	Original Size	**Increase in Size**	Original Size	Increase in Size
Branches	1	**49,425**	1	**166**	1	142
Tellers	6	**18,080**	5	**1,021**	5	171
Accounts	155,173	**243,193**	147,541	**245,835**	147,541	5,523

Multiple autovaccum processes helped small tables, but not large tables

EnterpriseDB™

Postgres 8.3 – HOT (Retail Vacuum)

	Postgres 8.2		Postgres 8.3 Pre HOT		Postgres 8.3 Post HOT	
	Original Size	**Increase in Size**	Original Size	**Increase in Size**	Original Size	**Increase in Size**
Branches	1	**49,425**	1	**166**	1	**142**
Tellers	6	**18,080**	5	**1,021**	5	**171**
Accounts	155,173	**243,193**	147,541	**245,835**	147,541	**5,523**

EnterpriseDB™

Several Ideas

- Update In Place
  - The first design. Replace old version with the new version and move old version somewhere else
  - It was just too complicated!
- Heap Overflow Tuple
  - That's what HOT used to stand for
  - A separate overflow relation to store the old versions.
  - Later changed so that the new version goes into the overflow relation and pulled into the main relation when old version becomes dead.
  - Managing overflow relation and moving tuples around was painful.
- Heap Only Tuple
  - That's what HOT stands for today
  - Tuples without index pointers

HOT Update

Necessary Condition A: *UPDATE does not change any of the index keys*

Example:

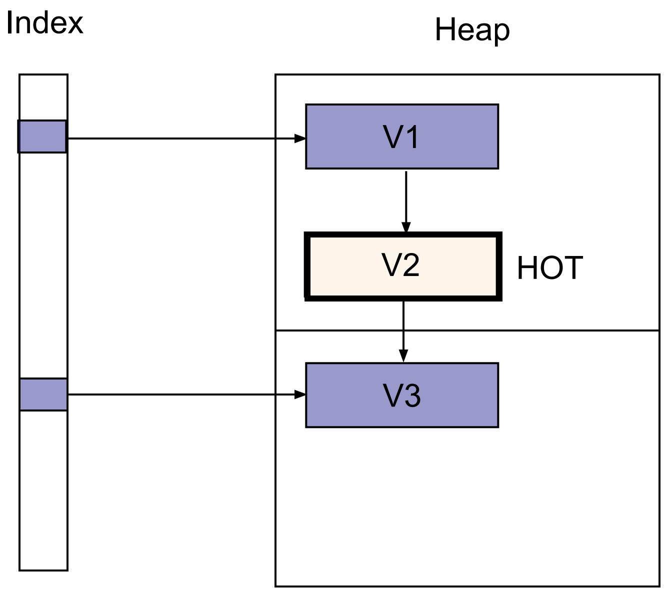
```
CREATE TABLE test (a int, b char(20));
CREATE UNIQUE INDEX textindx ON test(a);
INSERT INTO test VALUES (1, ‘foo’);

UPDATE test SET b = ‘bar’ WHERE a = 1;
UPDATE test SET a = a + 1 WHERE a = 1;
```

First UPDATE changes the non-index column – candidate for HOT update

Second UPDATE changes the index column – HOT update not possible

HOT Update



- V1 is updated – no index key change
  Single Index Entry Update Chain

- V2 is updated – no free space in block

Necessary Condition B*: The new version should fit in the same old block – HOT chains can not cross block boundary.*

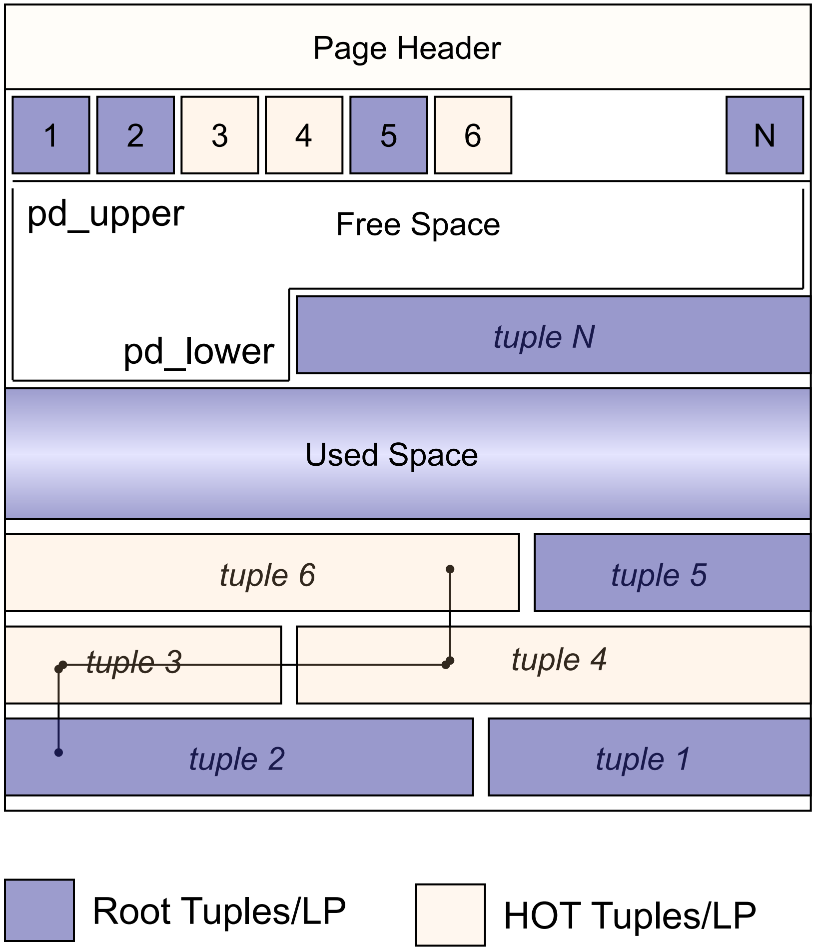
HOT Update – Necessary Conditions

Necessary Condition A: *UPDATE does not change any of the index keys*

Necessary Condition B*: The new version should fit in the same old block – HOT chains can not cross block boundary.*

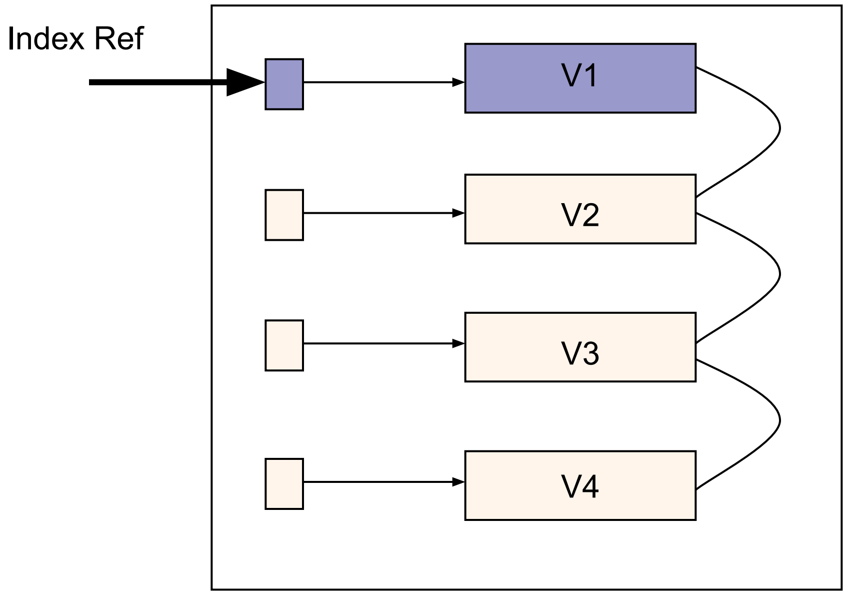
EnterpriseDB™

Inside A Block



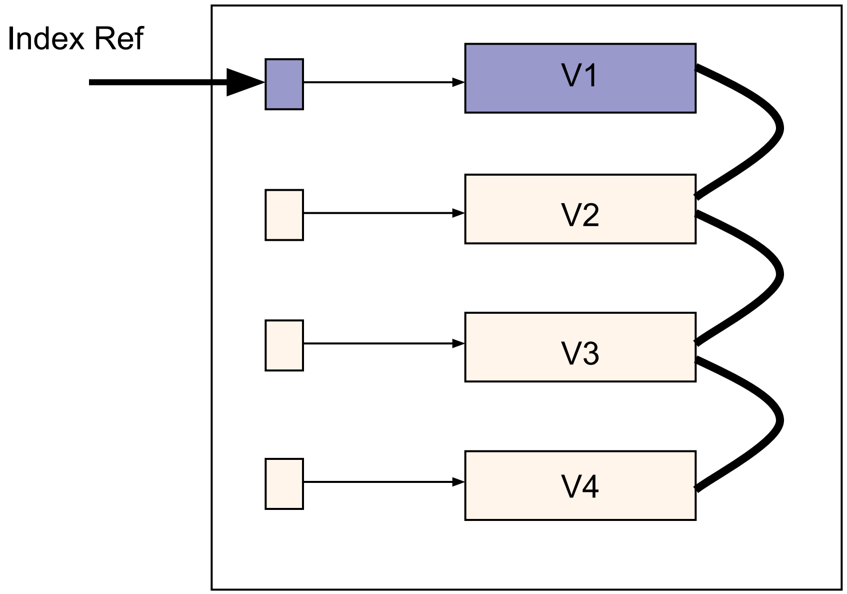
- Page Header followed by line pointers
- Line pointers point to the actual tuples
- Indexes always point to the line pointers and not to the actual tuple
- HOT chains originate at Root LP and may have one or more HOT tuples
- **HOT tuples are not referenced by the indexes directly.**

HOT – Heap Scan



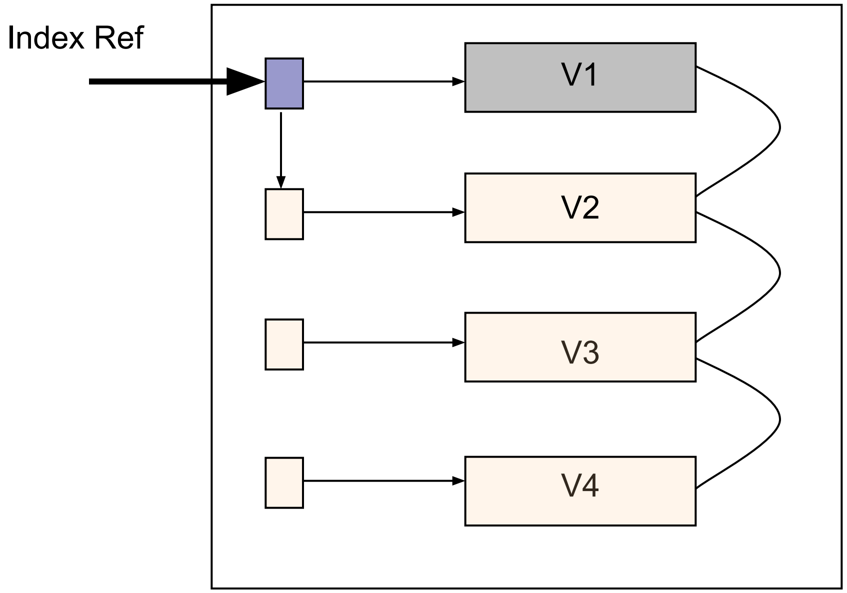
- No change to Heap Scan
- Each tuple is examined separately and sequentially to check if it satisfies the transaction snapshot

HOT – Index Scan



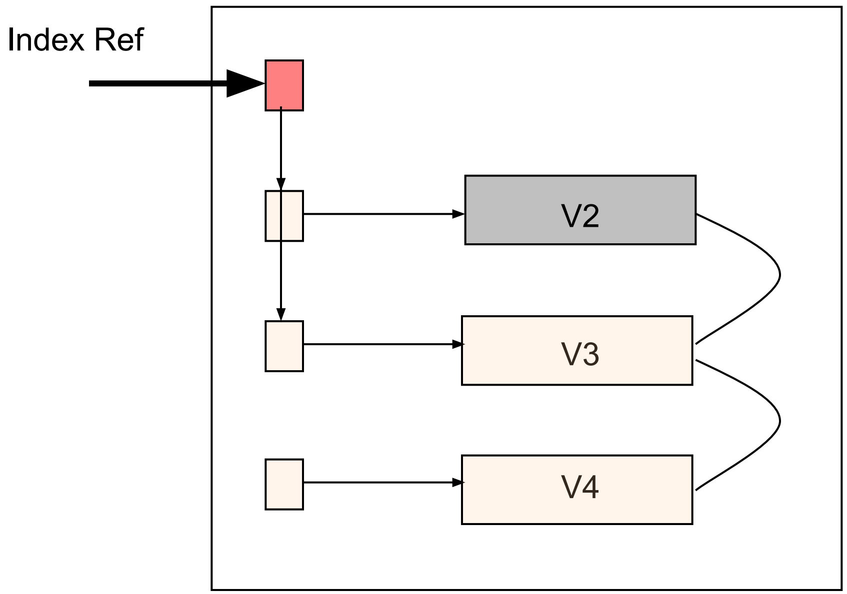
- Index points to the Root Tuple
- If the Root tuple does not satisfy the snapshot, the next tuple in the HOT chain is checked.
- Continue till end of the HOT chain
- The Root tuple can not be removed even if it becomes DEAD because index scan needs it

Pruning – Shortening the HOT Chain



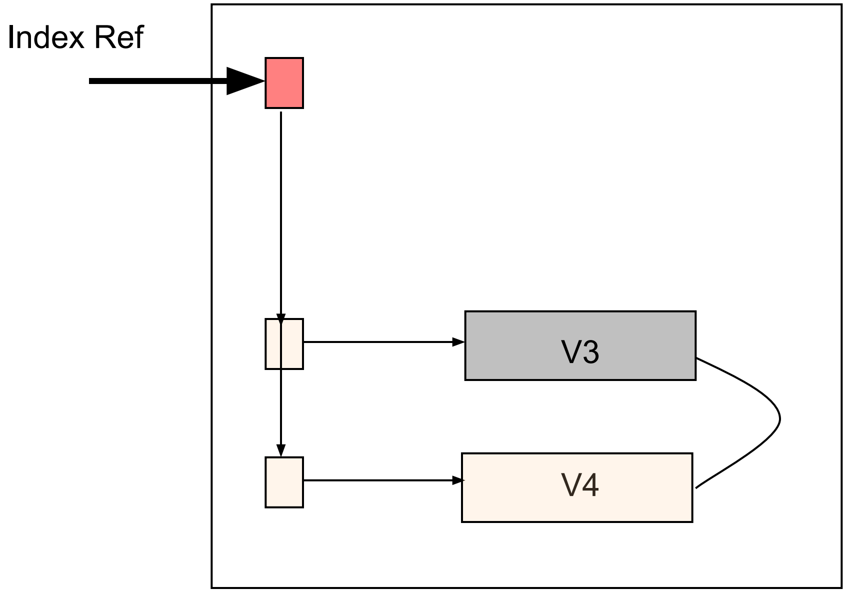
- V1 becomes DEAD
- V1 is removed, but it’s line pointer (LP) can not be removed – index points to it
- Root LP is redirected to the LP of next tuple in the chain

Pruning – Shortening the HOT Chain



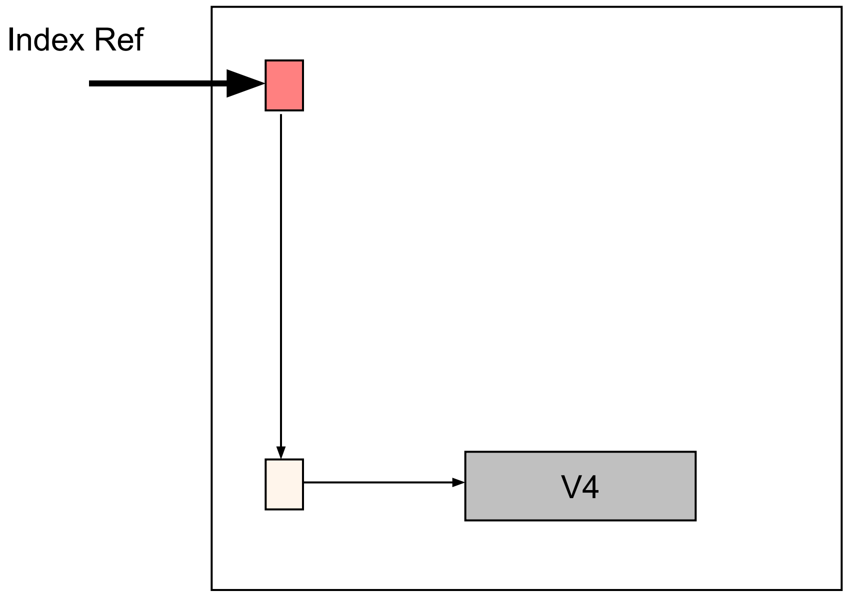
- Root LP is a redirected LP
- V2 becomes DEAD
- V2 and it's LP is removed – HOT tuple
- Root LP now redirects to the next tuple in the chain

Pruning – Shortening the HOT Chain



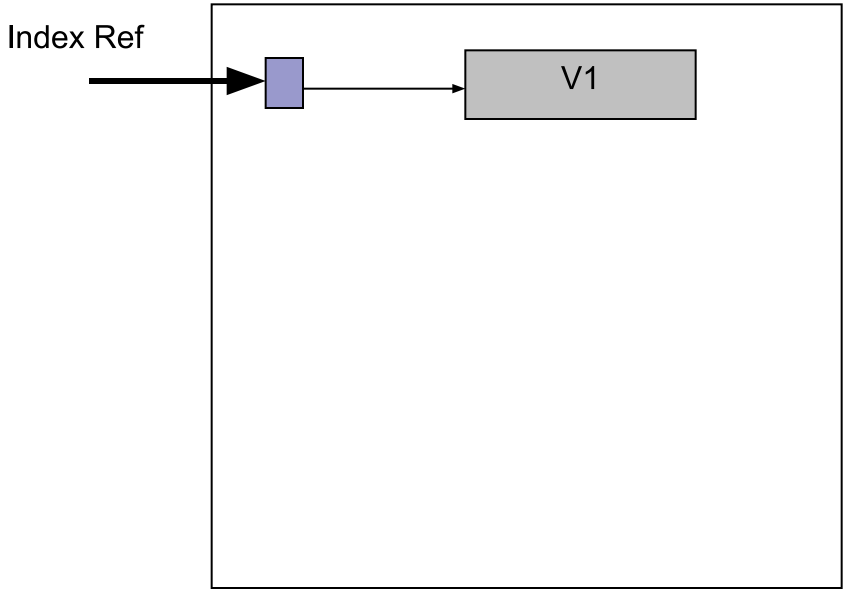
- Root LP is a redirected LP
- V3 becomes DEAD
- V3 and it’s LP is removed – HOT tuple
- Root LP now redirects to the next tuple in the chain

Pruning – Shortening the HOT Chain



- Root LP is a redirected LP
- V4 becomes DEAD
- V4 and it's LP is removed – HOT tuple
- Root LP is now DEAD – still can't be removed

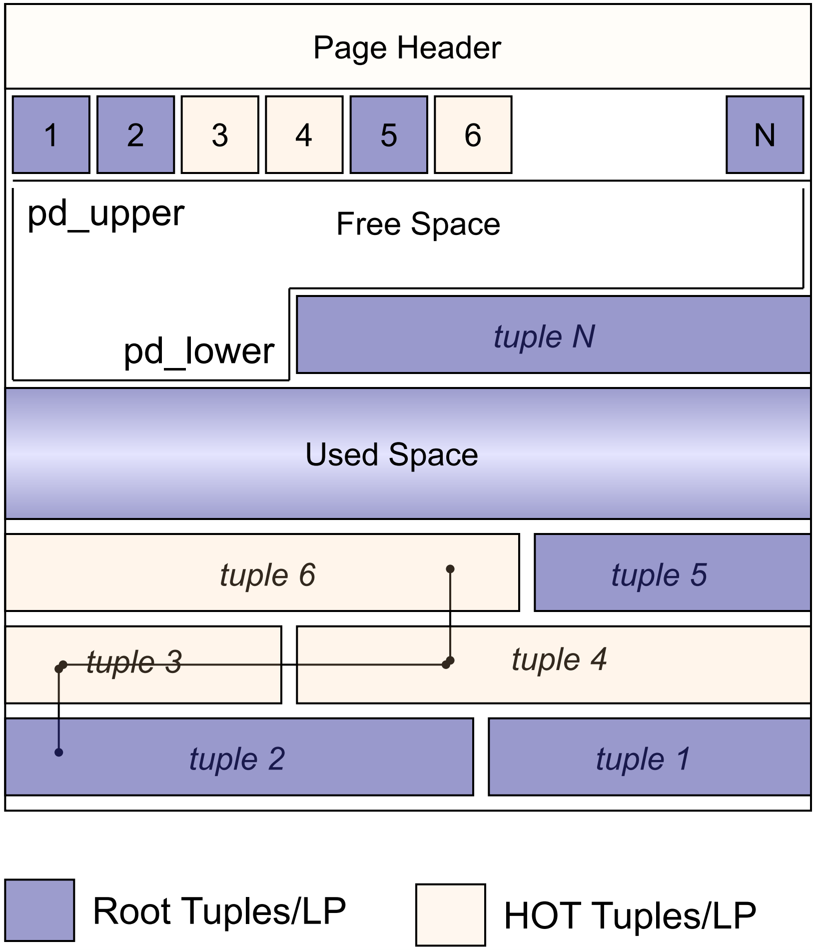
Pruning – Normal UPDATEs and DELETEs



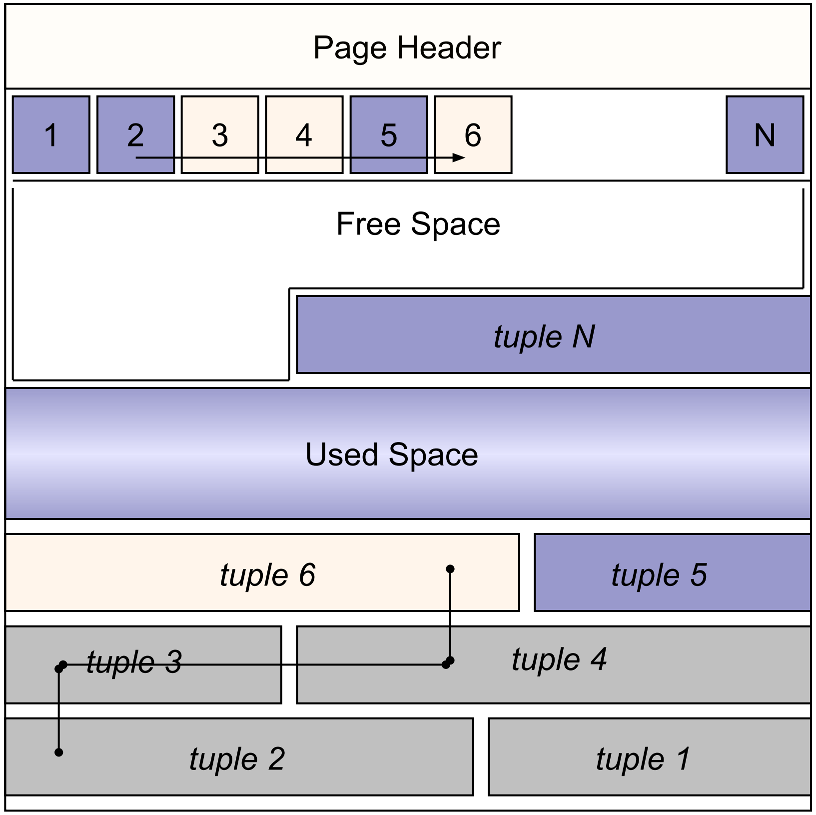
- Normal UPDATEd and DELETEd tuples are removed and their LPs are marked DEAD – LPs can't be removed
- A very useful side-effect of HOT

EnterpriseDB™

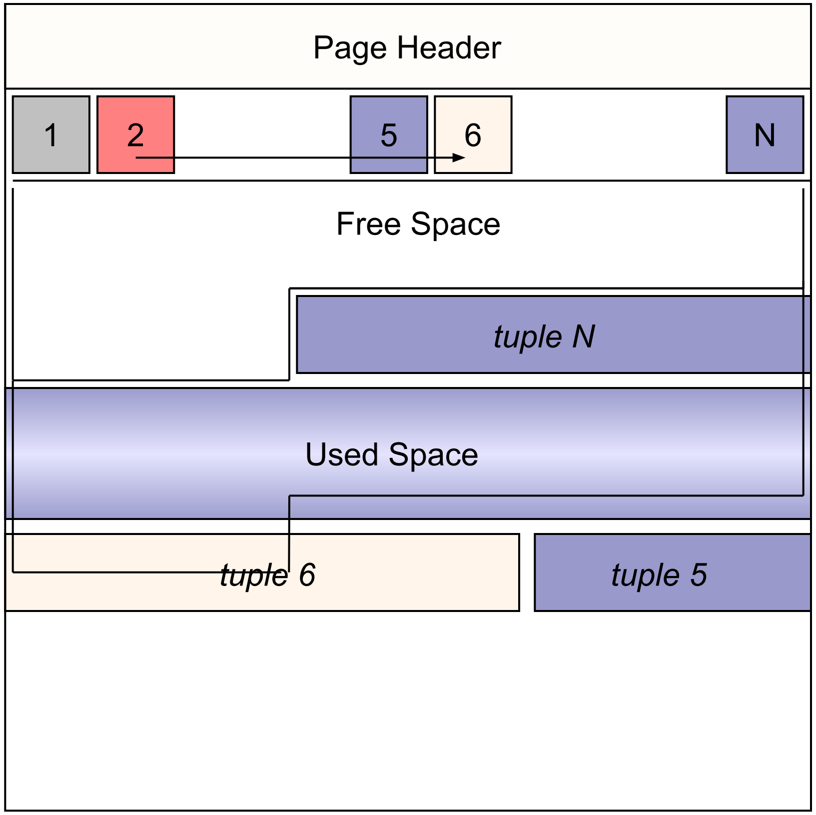
Pruning and Defragmentation



Pruning – Recovering Dead Space



Defragmentation – Collecting Dead Space



Billion $ Question – When to Prune/Defragment ?

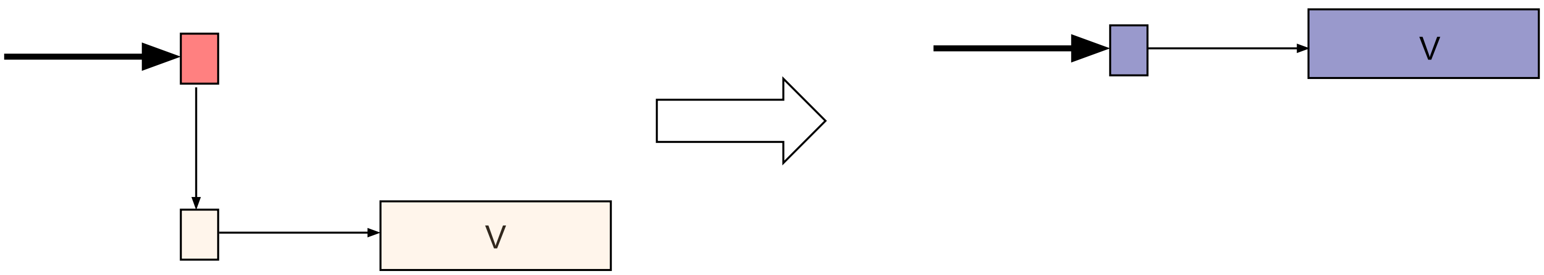
- Pruning and defragmentation (PD) happens together – requires cleanup lock on the buffer and shuffles tuples in a page.
- Too frequent PD may conflict with other backends accessing the buffer.
- Too infrequent PD may slow down reclaiming dead space and create long HOT chains.
- Page level hint bits and transaction id is used to optimize PD operations.

Page Level Hints and Xid

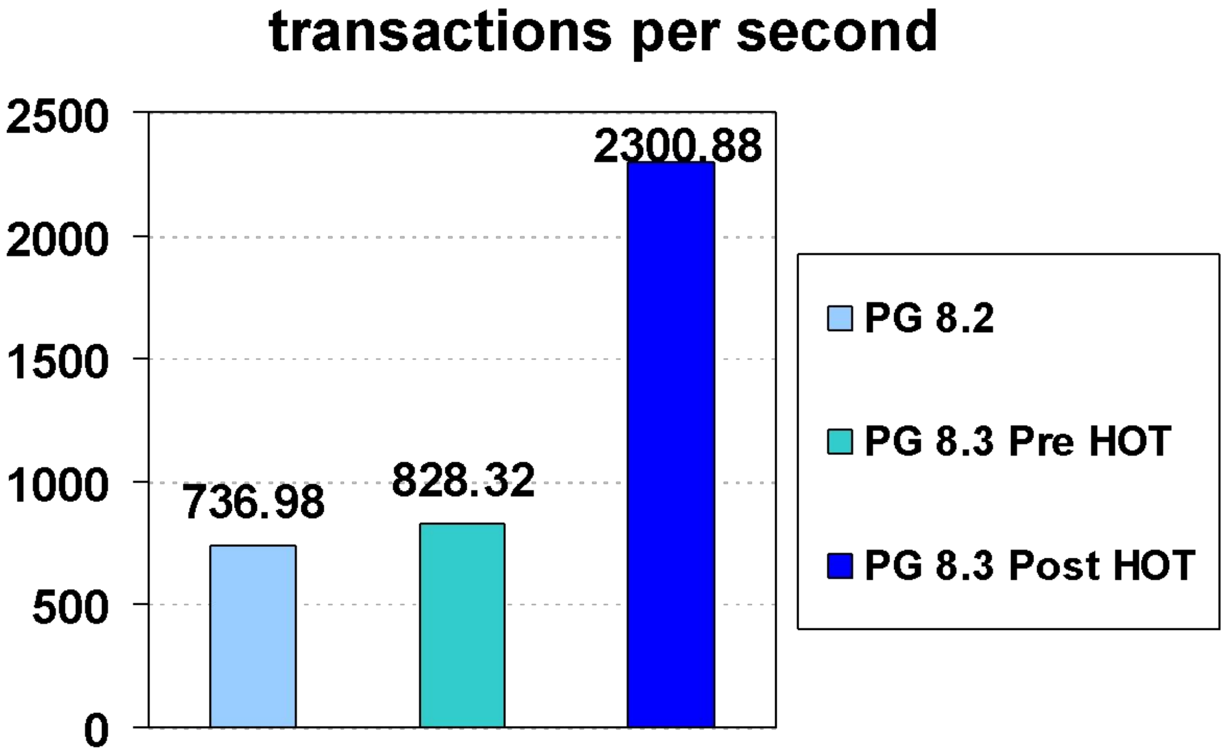
- If UPDATE does not find enough free space in a page, it does COLD UPDATE but sets PD_PAGE_FULL flag
- The next access to page may trigger prune/defrag operation if cleanup lock is available.
- PD never waits for cleanup lock
- Page Xid is set to the oldest transaction id which deleted or updated a tuple in the page. PD is usable only if RecentGlobalXmin is less than the Page Xid.

Lazy Vacuum / Vacuum Full

- Lazy Vacuum is almost unchanged.
- DEAD line pointers are collected and reclaimed.
- Vacuum Full clears the redirected line pointers by making them directly point to the first visible tuple in the chain.



Headline Numbers - Comparing TPS



That's a good 200% increase in TPS

Comparing Heap Bloat (# blocks)

	Postgres 8.2		**Postgres 8.3 Pre HOT**		**Postgres 8.3 Post HOT**	
	Original Size	**Increase in Size**	Original Size	**Increase in Size**	Original Size	**Increase in Size**
Branches	1	**49,425**	1	**166**	1	**142**
Tellers	6	**18,080**	5	**1,021**	5	**171**
Accounts	155,173	**243,193**	147,541	**245,835**	147,541	**5,523**

HOT significantly reduces heap bloat; for small and large tables

EnterpriseDB™

Comparing Index Bloat (# blocks)

	Postgres 8.2		Postgres 8.3 Pre HOT		Postgres 8.3 Post HOT	
	Original Size	**Increase in Size**	Original Size	**Increase in Size**	Original Size	**Increase in Size**
Branches	2	**1,023**	2	**588**	2	**4**
Tellers	5	**353**	5	**586**	5	**19**
Accounts	24,680	**24,679**	24,680	**24,677**	24,680	**0**

HOT significantly reduces index bloat too; for small and large tables

EnterpriseDB™

Comparing IO Stats

		Postgres 8.2		Postgres 8.3 Pre HOT		Postgres 8.3 Post HOT	
		Blks Read	Blks Hit	Blks Read	Blks Hit	Blks Read	Blks Hit
Branches	H	576,949	810,688,147	8,595	2,904,470,056	784	74,640,567
	I	7,540	330,165,668	68,992	254,298,111	7	56,184,941
Tellers	H	685,599	219,033,182	7,710	452,528,173	678	62,275,473
	I	366	135,684,700	599	210,984,757	28	60,655,207
Accounts	H	20,138,195	167,902,036	19,065,032	173,465,111	162,867	101,354,726
	I	464,641	266,747,533	482,835	270,662,463	49,327	181,307,038

Comparing IO Stats

		Postgres 8.2		Postgres 8.3 Pre HOT		Postgres 8.3 Post HOT	
		Blks Read	Blks Hit	Blks Read	Blks Hit	Blks Read	Blks Hit
Branches	**H**	**576,949**	**810,688,147**	**8,595**	**2,904,470,056**	**784**	**74,640,567**
	I	7,540	330,165,668	68,992	254,298,111	7	56,184,941
Tellers	H	685,599	219,033,182	7,710	452,528,173	678	62,275,473
	I	366	135,684,700	599	210,984,757	28	60,655,207
Accounts	H	20,138,195	167,902,036	19,065,032	173,465,111	162,867	101,354,726
	I	464,641	266,747,533	482,835	270,662,463	49,327	181,307,038

Comparing IO Stats

		Postgres 8.2		Postgres 8.3 Pre HOT		Postgres 8.3 Post HOT	
		Blks Read	Blks Hit	Blks Read	Blks Hit	Blks Read	Blks Hit
Branches	H	576,949	810,688,147	8,595	2,904,470,056	784	74,640,567
	I	7,540	330,165,668	68,992	254,298,111	7	56,184,941
Tellers	H	685,599	219,033,182	7,710	452,528,173	678	62,275,473
	I	366	135,684,700	599	210,984,757	28	60,655,207
Accounts	**H**	**20,138,195**	**167,902,036**	**19,065,032**	**173,465,111**	**162,867**	**101,354,726**
	I	464,641	266,747,533	482,835	270,662,463	49,327	181,307,038

Significant reduction in IO improves the headline numbers

EnterpriseDB™

What Should I Do ?

- Nothing! HOT is always enabled and there is no way to disable it.
- It works on user and system tables
- A heap fill factor less than 100 may help
- A significantly smaller heap fill factor (as low as 50) is useful for heavy updates where most of the updates are bulk updates
- Non index key updates is a necessary condition for HOT – check if you don't need one of the indexes.
- Prune-defrag reclaims COLD UPDATEd and DELETEd DEAD tuples by converting their line pointers to DEAD
- You still need VACUUM – may be less aggressive

Limitations

- Free space released by defragmentation can only be used for subsequent UPDATEs in the same page – we don't update FSM after prune-defragmentation
- HOT chains can not cross block boundaries
- Newly created index may remain unusable for concurrent transactions
- Normal vacuum can not clean redirected line pointers

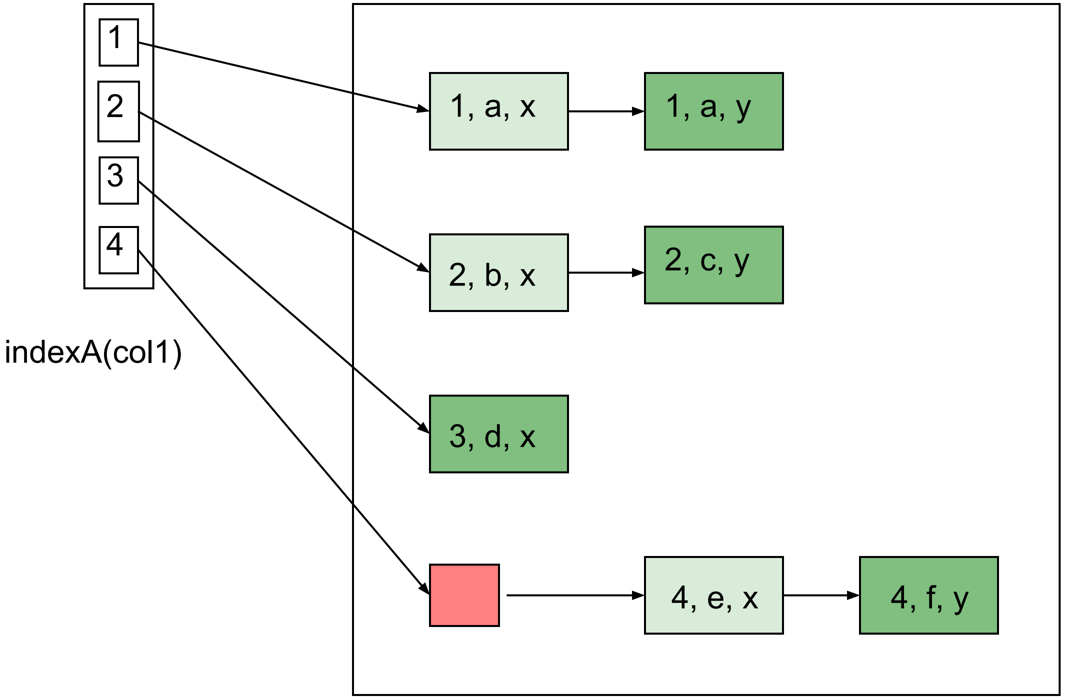
Create Index

- This was one of the most interesting challenges in HOT development.
- The goal was to support CREATE INDEX without much or no impact on the existing semantics.
- Did we succeed ? Well, almost

Create Index - Challenges

- Handling broken HOT chains
- New Index must satisfy HOT properties
  - All tuples in a HOT chain must share the same index key
  - Index should not directly point to a HOT tuple.
- Create Index should work with a ShareLock on the relation

Create Index – Sane State

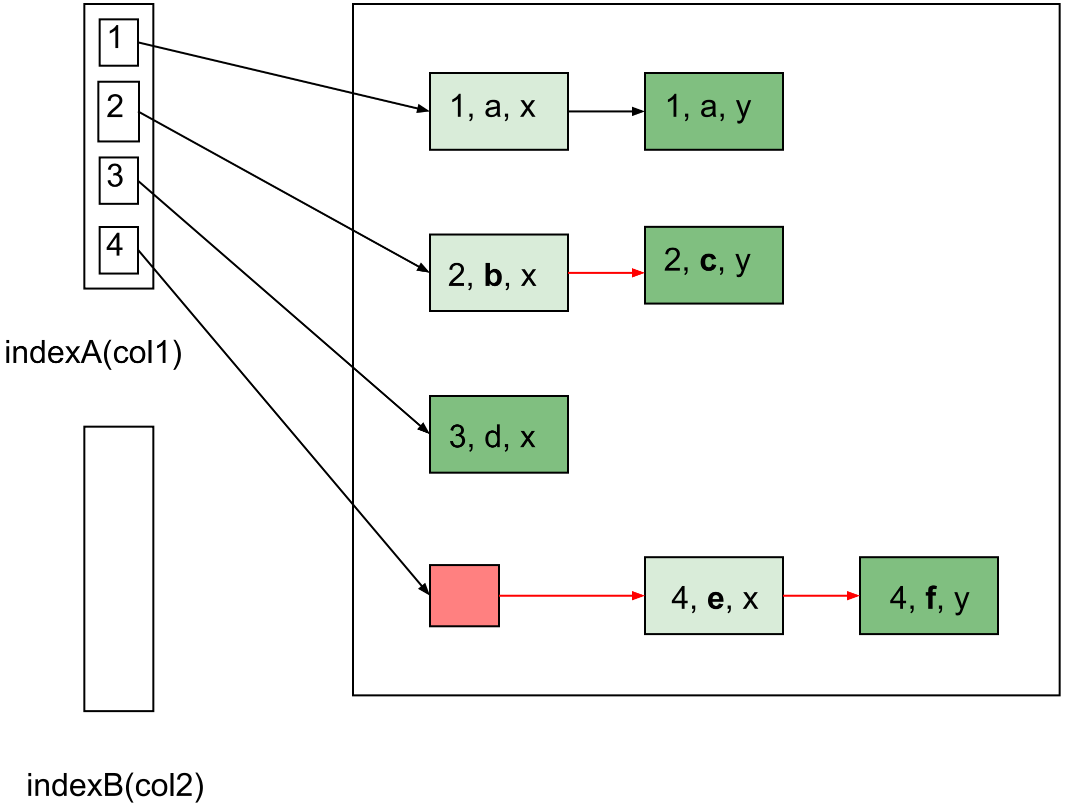


- All HOT chains are in sane state
- Every tuple in a chain shares the same index key
- Index points to the Root Line Pointer

Create Table test (col1 int, col2 char, col3 char);

Create Index indexA ON test(col1);

Create Index – Broken HOT Chains

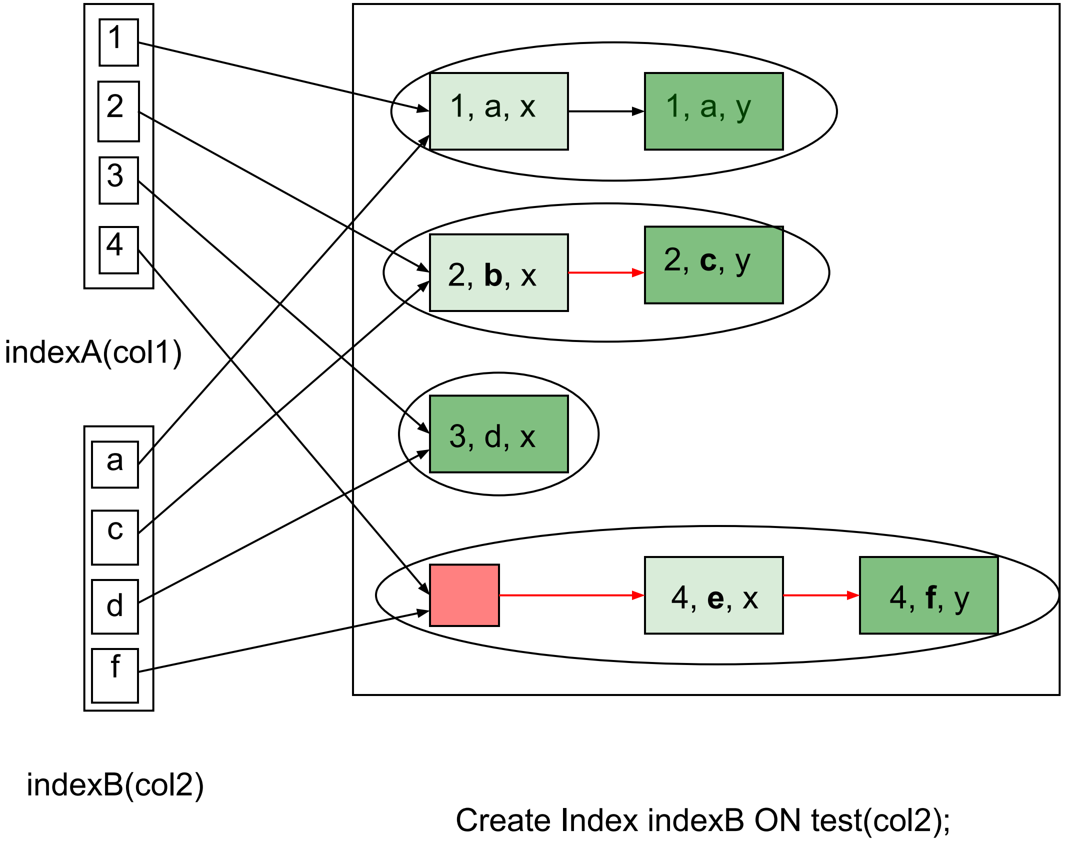


- **Create a new Index on col2**
- Second and fourth HOT chains, marked with → , are broken w. r. t. new Index
- tuples are recently dead, but may be visible to concurrent txns

Create Index indexB ON test(col2);

EnterpriseDB™

Create Index – Building Index with Broken HOT Chains



- Recently Dead tuples are not indexed
- Index remains unusable to the transactions which can potentially see these skipped tuples, including the transaction which creates the index
- Any new transaction can use the index
- xmin of pg_class row is used to check index visibility for transactions

Thank you

pavan.deolasee@gmail.com
pavan.deolasee@enterprisedb.com