Pagination Done the PostgreSQL Way
About Me
Note

In this presentation index means B-tree index.
A Trivial Example

A query to fetch the 10 most recent news:

```sql
select *
from news
where topic = 1234
order by date desc, id desc
limit 10;

create index .. on news(topic);
```

Using `order by` to get the most recent first and `limit` to fetch only the first 10.

Alternative SQL-2008 syntax (since PostgreSQL 8.4)
fetch first 10 rows only
Worst Case: No Index for order by

Limit (actual rows=10)  
-> Sort (actual rows=10)  
  Sort Method: top-N heapsort  Memory: 18kB  
-> Bitmap Heap Scan (rows=10000)  
  Recheck Cond: (topic = 1234)  
-> Bitmap Index Scan (rows=10000)  
  Index Cond: (topic = 1234)
Worst Case: No Index for order by

The limiting factor is the number of rows that match the where clause (Base-Set Size).

The database might use an index to satisfy the where clause, but must still fetch all matching rows to “sort” them.
Another Benchmark: Fetch Next Page

Fetching the next page is easy using the offset keyword:

```sql
select *
from news
where topic = 1234
order by date desc, id desc
offset 10
limit 10;
```
Worst Case: No Index for order by

Limit (actual rows=10)
-> Sort (actual rows=20)
  Sort Method: top-N heapsort  Memory: 19kB
-> Bitmap Heap Scan (actual rows=10000)
  Recheck Cond: (topic = 1234)
-> Bitmap Index Scan (actual rows=10000)
  Index Cond: (topic = 1234)
Worst Case: No Index for order by

Sorting might become the limiting factor when browsing farther back.

Fetching the last page can take considerably longer than fetching the first page.
Improvement 1: Indexed order by

```
select *
from news
where topic = 1234
order by date desc, id desc
offset 10
limit 10;

create index .. on news (topic, date, id);
```

A single index to support the where and order by clauses.
Improvement 1: Indexed order by

Limit (actual rows=10)
-> Index Scan Backward (actual rows=20)
  Index Cond: (topic = 0)
Improvement 1: Indexed order by

Fetching the first page is not affected by the Base-Set size!

Fetching the next page is also faster. However, PostgreSQL might take a Bitmap Index Scan when browsing to the end.
We can do better!
Don’t touch what you don’t need
Improvement 2: The Seek Method

Instead of offset, use a `WHERE` filter to remove the rows from previous pages.

```sql
select *
from news
where topic = 1234
    and (date, id) < (prev_date, prev_id)
order by date desc, id desc
limit 10;
```

Only select the rows “before” (=earlier date, id) the last row from the previous page.

A definite sort order is **really** required!
Side Note: Row Values/Constructors

Besides scalar values, SQL also defines “row values” or “composite values.”

- In the SQL standard since ages (SQL-92)
- All comparison operators are well defined
  - E.g.: 
    
    \[(x, y) > (a, b) \text{ is true iff } (x > a \text{ or } (x=a \text{ and } y>b))\]
  - In other words, \textit{when} \((x,y)\) sorts after \((a,b)\)
- Great PostgreSQL support since 8.0!
Seek Method w/o Index for order by

Limit (actual rows=10)
-> Sort (actual rows=10)
  Sort Method: top-N heapsort  Memory: 18kB
-> Bitmap Heap Scan (actual rows=10)
  Rows Removed by Filter: 10 (new in 9.2)
-> Bitmap Index Scan (actual rows=10000)
  Index Cond: (topic = 1234)
Seek Method w/o Index for order by

Always needs to retrieve the full base set, but the top-n sort buffer needs to hold only 10 rows.

The response time remains constant even when browsing to the last page. And the memory footprint is very low!
Seek Method with Index for order by

Limit (\texttt{actual rows=10})
-> Index Scan Backward (\texttt{actual rows=10})
Index Cond: \((\text{topic} = 1234) \land (\text{ROW}(dt, id) < \text{ROW}(\ldots, 12345)))\)
Seek Method with Index for order by

Successively browsing back doesn’t slow down.

Neither the size of the base set(*) nor the fetched page number affects the response time.

(*) the index tree depth still affects the response time.
Comparison

W/O index for order by

With index for order by
Too good to be true?

The Seek Method has serious limitations

- You cannot **directly** navigate to arbitrary pages
  - because you need the values from the previous page

- Bi-directional navigation is possible but tedious
  - you need to reverse the order by direction and RV comparison

- Works best with full row values support
  - Workaround is possible, but ugly and less performant
  - Framework support?
A Perfect Match for Infinite Scrolling

The “Infinite Scrolling” UI doesn’t need to ...

- navigate to arbitrary pages
  - there are no buttons

- Browse backwards
  - previous pages are still in the browser

- show total hits
  - if you need to, you are doomed anyway!
Also a Perfect Match for PostgreSQL

### Row Values Support Matrix

<table>
<thead>
<tr>
<th>Feature</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQLite</th>
<th>SQL Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supported in where clause</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Ranges supported ((&lt;,&gt;)</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Optimal index usage</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

### Order by Support Matrix

<table>
<thead>
<tr>
<th>Feature</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQLite</th>
<th>SQL Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read index backwards</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Order by ASC/DESC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Index ASC/DESC</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Order by NULLS FIRST/LAST</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Default NULLS order</td>
<td>First</td>
<td>Last</td>
<td>Last</td>
<td>First</td>
<td>x</td>
</tr>
<tr>
<td>Index NULLS FIRST/LAST</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>