Swarm64DB Feature Documentation

When to use Swarm64DB?

Swarm64DB is offers its best performance if any one, or better yet, several of these conditions are met:

- inserting data at high velocity (e.g. one million rows/sec)
- when looking for specific information or aggregating ranges in very large datasets (e.g. looking for a price range or analysing a date range)
- real-time requirements (sub-second timing between new information received by the database and availability of this data for query)
- when fast indexing and compact storage of large datasets is important
- when already using PostgreSQL, MySQL or MariaDB and limited by available performance
- when working in an environment with limited I/O bandwidth and high I/O latency (e.g. cloud instances without local storage)

How does Swarm64DB work?

Swarm64DB plugs into PostgreSQL via the Foreign Data Wrapper and into MariaDB or MySQL via the Storage Engine interface. Swarm64DB uses both the CPU and the FPGA to share the workload during data ingestion and analytics. The FPGA executes data compression and decompression as well as filtering and data pre-processing steps from the SQL query. The CPU focuses on planning which parts of the query are offloaded to the FPGA, executing the remainder of the query and maintains the “Optimized Columns”, Swarm64DB’s OLAP optimized equivalent of an index.
The CPU and FPGA cooperate using memory mapped files that are coherent between multiple database workers executing in parallel. Memory mapping avoids double-buffering (storing data twice, both in the operating system I/O cache and the database cache).

**How do the Swarm64DB Optimized Columns work?**

Swarm64 employs a proprietary mechanism to arrange the data for FPGA and database processing. Optimized Columns allow Swarm64 to very efficiently process frequent values or value ranges in any of these columns, in any order. Reading through these ranges is also highly optimized for I/O throughput, especially when low latency local storage is not available or desirable (cloud environments, scalable enterprise storage etc.).

Optimized Columns allow Swarm64DB to only access the data relevant to the query, as shown in the illustration below, which illustrates the concept of defining two Optimized Columns, ws_order_number and ws_sold_date, on the web_sales table. During the query, only the data that matches both SELECT ranges is processed by the CPU, the rest is filtered out before processing by a cooperation between the FPGA and the Optimized Columns data structure.

**Summary:**

Optimized Columns accelerate the WHERE conditions that are applied to them. They are Swarm64DB’s equivalent of an index but create less than 5% storage overhead and scale very well with big datasets.

**How best to use the Swarm64 Optimized Columns?**

The ranges on Optimized Columns need to be constant at the time the query executes. If the ranges are variable during execution, e.g., the conditions being the result of a complex sub-select, the Optimized Columns cannot apply their filter. When selecting only hundreds of rows or less, there is significant speed-up versus a native table scan but the Optimized Columns are somewhat slower than a standard B-Tree index.

As a result, Optimized Columns should be used where querying for frequent data points or data ranges is common. Good examples are time or date ranges, price ranges or a specific price point, quantities, tracking or sensor data, spatial information and many more. Unlike Multi-Part Keys in the native database, Optimized Columns can be queried independently from each other and in any order (order in which they are defined has no impact). The more Optimized Columns are queried, the faster the query executes. This behaviour is very different from the native Multi-Part Keys, which perform best if they are used in order and skipping earlier key parts carries a heavy performance penalty.
Up to 4 columns can be optimized, but with fewer performance increases. Swarm64 recommends using only one or two Optimized Columns more than the important queries select on. In a table most frequently queried for only one of the columns, up to two or three columns can be optimized. Conversely, Swarm64 only recommends using all four Optimized Columns if two of them are always queried in tandem (e.g., in a spatial query case, always asking for both the x and y axis). Swarm64 will significantly expand this capability in future releases.

As a rule of thumb, one should not use Optimized Columns on IDs or on columns used solely to join against other tables. Since Optimized Columns do not work like the native B-Tree indexes, applying the optimization will not accelerate the join the same way a native index may. The exception are ID ranges or join ranges that are consecutive, e.g., a date range represented by an ascending date ID.

**What performance can be expected (by query type)?**

Below are specific query examples based on the `web_sales` table layout from TPC-DS. Three Optimized Columns are defined, `ws_item_sk`, `ws_bill_customer_sk` and `ws_sold_date_sk`.

**Example table (Source - TPC-DS data set):**

```sql
CREATE FOREIGN TABLE web_sales
(
  ws_sold_date_sk INTEGER,
  ws_sold_time_sk INTEGER,
  ws_item_sk INTEGER NOT NULL,
  ws_bill_customer_sk INTEGER,
  [...] 
  ws_item_sk INTEGER,
  [...] 
  ws_bill_customer_sk INTEGER,
  [...] 
  ws_web_page_sk INTEGER,
  ws_web_site_sk INTEGER,
  ws_ship_mode_sk INTEGER,
  ws_warehouse_sk INTEGER,
  ws_order_number INTEGER NOT NULL,
  ws_quantity INTEGER,
  ws_wholesale_cost DECIMAL(7,2),
  [...] 
) SERVER swarm_server OPTIONS (optimized_columns 'ws_item_sk, ws_bill_customer_sk, ws_sold_date_sk');
```

Complexity of the aggregation (marked in grey in the examples below) has some effect on speed. Very complex aggregations may show less speed gain versus the native database because the last steps of the complex aggregation mechanism run inside the native database and therefore at the regular speed. WHERE conditions are shown framed (Optimized Columns) or framed and red (other columns).

All examples are based on large datasets (10 GB and larger). Query acceleration is descending, the topmost perform best.

<table>
<thead>
<tr>
<th>Query</th>
<th>Acceleration</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>INSERT INTO web_sales VALUES (value1, value2, value3, [...]</code>);</td>
<td>2.3x-10.6x faster inserts with the same amount of optimized/indexed columns – measured against MariaDB and PostgreSQL (with their settings optimized for ingestion, such as relaxed rules for flushing to storage).</td>
</tr>
</tbody>
</table>
### Selecting one or multiple ranges on a single table using the Optimized Columns:

Very high acceleration factors measured versus an similarly indexed PostgreSQL or MariaDB *(expect > 9.2x)*

```
SELECT ws_ship_mode_sk as Shipping_Mode,  
SUM(ws_quantity) as Total_Sold,  
AVG(ws_wholesale_cost) as Average_Wholesale_Cost  
FROM web_sales  
WHERE  
ws_item_sk BETWEEN 70526 AND 93102 AND  
ws_sold_date_sk > 2452000  
GROUP BY Shipping_Mode  
ORDER BY Total_Sold DESC;
```

### Selecting one or multiple ranges on a single table using both optimized and regular columns (red):

High acceleration factors measured versus an similarly indexed PostgreSQL or MariaDB *(expect ~ 9.2x)*

```
SELECT [...] FROM web_sales  
WHERE  
ws_item_sk BETWEEN 70526 AND 93102 AND  
ws_ship_date_sk > 2452000  
GROUP BY [...];
```

### Selecting one or multiple ranges on a single table using only regular columns (red):

Expect 2.1x acceleration factor measured versus PostgreSQL, 5.3x versus MariaDB

```
SELECT [...] FROM web_sales  
WHERE  
ws_sold_time_sk BETWEEN 2800 AND 54000  
AND ws_ship_date_sk > 2452000  
GROUP BY [...];
```

### Select on a condition which changes during query run-time OR operating on the entire table without any limiting condition:

Versus typical network attached storage or cloud environments: **up to 4.5x** acceleration due to HW-accelerated (de)compression and the ability to hide I/O latency

Versus high-end local memory (e.g. an array of PCIe based NVMEs): no acceleration

```
SELECT (2017 - c_birth_year) as Customer_Age_Group,  
SUM(ws_quantity) as Total_Sold,  
AVG(ws_wholesale_cost) as Average_Wholesale_Cost  
FROM web_sales  
INNER JOIN customer  
ON ws_bill_customer_sk=c_customer_sk  
WHERE  
ws_item_sk BETWEEN 70526 AND 93102 AND  
ws_sold_date_sk > 2452000  
GROUP BY c_birth_year;
```

### Conditions or GROUP BYs influenced by a JOIN:

On unindexed columns **up to 2.1x** faster than native database.

**Slow-downs are possible when measured against index joins in the native database** (index join – joining two tables using indexed columns). This is case dependent, if joining against ranges (e.g. date range) Optimized Columns can outperform the native index.

**Solutions when encountering a slow join:**

- Where applicable, pre-join the column(s) from the WHERE conditions into the larger table, then join the remaining columns into the result set. Swarm64’s compression will minimize the storage impact of this

- Keep join-heavy operations in correctly indexed native database tables and migrate to Swarm64DB later, hardware accelerated JOINs are on the roadmap
Summary
For best performance, use WHERE conditions with constant ranges during the query. They perform best on the Optimized Columns. If possible, apply these WHERE conditions to all big tables in the query. Number of rows matched by the condition has a major effect on performance – for large numbers of matching rows execution time slows from more than 9.2\texttimes speed up (when less than 10\% of the rows are matched) to 2.1\texttimes (100\% matched, analogous to a full table scan).

What performance can be expected working with Spatial Data?
Swarm64DB performs particularly well when working with big datasets of spatial data, especially when combining spatial data with other payloads (temperature sensor readout etc.).

Example table – IoT case:
CREATE FOREIGN TABLE device_location_data
(
    device_id INTEGER,
    geo_latitude INTEGER NOT NULL,
    geo_longitude INTEGER NOT NULL,
    orientation_angle INTEGER NOT NULL, -- in degrees, north = 0°
    temperature INTEGER NOT NULL
) SERVER swarm_server OPTIONS (optimized_columns 'geo_latitude, geo_longitude, orientation_angle, temperature');

Performance by Query Type:

<table>
<thead>
<tr>
<th>Query</th>
<th>Acceleration</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT device_id, temperature FROM device_location_data WHERE geo_latitude BETWEEN 52452264 AND 52455631 AND geo_longitude BETWEEN 13376830 AND 13389387;</td>
<td>In simple spatial queries, such as checking which devices are within a bounding box, Swarm64DB’s Optimized Columns shine (expect &gt;100\texttimes speed-up vs. table scan in the native database)</td>
</tr>
<tr>
<td>SELECT device_id FROM device_location_data WHERE geo_latitude BETWEEN 52452264 AND 52455631 AND geo_longitude BETWEEN 13376830 AND 13389387 AND orientation_angle BETWEEN -5 AND 20 AND temperature &gt; 10;</td>
<td>Unlike existing spatial indexes, Swarm64DB can combine spatial information (latitude, longitude, orientation angle) with independent data (e.g. temperature) and deliver even faster results, especially when filtering on all Optimized Columns at the same time.</td>
</tr>
</tbody>
</table>

These examples illustrate working with plain sensor read-outs (e.g. 4 byte integer). It is also possible to use the geometric custom types and benefit from the rich available feature-set (such as PostGis for PostgreSQL).
## Swarm64DB: Supported Native Database Versions

<table>
<thead>
<tr>
<th>Swarm64DB</th>
<th>PostgreSQL</th>
<th>MariaDB</th>
<th>MySQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0.9, v1.0 compatible with:</td>
<td>9.5 and 9.6</td>
<td>10.1</td>
<td>5.7</td>
</tr>
<tr>
<td></td>
<td>PostgreSQL recommended, talk to Swarm64 if MariaDB is preferable. Swarm64 can test compatibility with the specific version required.</td>
<td>PostgreSQL recommended, talk to Swarm64 if MySQL is preferable. Swarm64 can test compatibility with the specific version required.</td>
<td></td>
</tr>
</tbody>
</table>