Surrogate Keys vs. Composite Keys in a Retail Fact Table: A Comparative Benchmark

Abstract

We compared two physical designs for a retail fact table—one clustered on a **composite key (CK)** and one built around a **surrogate key (SK)** with secondary indexes—using MySQL 8 in Docker. We ran the four business deliverables (spend trends, demographic influence, growth categories, and campaign impact) and six operational checks (household lookup, store+day micro timing, insert throughput, delete window, storage, and table maintenance). SK won **three of the four business queries**, especially where the workload is join/aggregate heavy. CK remained best for access aligned to (store, day) and was materially leaner to load and maintain. These results match InnoDB's clustered vs. secondary index behavior, EXPLAIN ANALYZE plans, and buffer-pool counters.[1–5]

---

1. Environment and data

- **Engine.** MySQL 8 (container sk_mysql), host port 3307 → container 3306
- **Database / user.** retail_db, user bench/benchpw
- **Client.** Jupyter (PyMySQL) for timing; MySQL Workbench for EXPLAIN ANALYZE
- **Data.** Base feed bench_lines_data (~1.42M rows) into both designs; dimensions: household_dim, product_dim, coupon_redemption_fact
- **Window used for business queries.** store_sk = 140, day BETWEEN 6 AND 36

---

2. Table designs

Composite key (CK)

bench_lines_ck with a clustered **PRIMARY KEY** on

(store_sk, day, product_sk, household_sk) and **no** secondary indexes.

Because the InnoDB primary key is clustered, rows are stored in this order; range filters on the leftmost columns scan tightly and sequentially.[1,2]

Surrogate key (SK)

bench_lines_sk with **PRIMARY KEY (id)** and two secondaries:

- idx_store_day_prod (store_sk, day, product_sk)
- idx_household (household_sk)

This decouples storage order from access paths and enables fast non-aligned lookups via secondary indexes.[1],[2]

---

3. Methods

We executed identical SQL for CK and SK variants, recorded elapsed time (ms) and rowcount, and checked equivalence by matching rowcounts. We used EXPLAIN ANALYZE to confirm chosen indexes and to read operator timings.[3] Buffer-pool counters were sampled before and after the heavier queries to observe logical/disk reads and read-ahead effects.[4]

---

4. Results: business deliverables (store 140, days 6..36)

Executive Summary — 4 Deliverables (Store 140, Days 6..36)

Task	CK (ms)	SK (ms)	Rows	Faster
spend_trends	5.47	6.63	177	**CK**
demo_influence	563.75	488.05	10	**SK**
growth_categories	1561.09	1247.66	10	**SK**
campaign_impact	6744.19	6119.41	2	**SK**

Summary: SK wins **3/4** deliverables (joins/aggregations); CK wins **spend_trends** because the query is perfectly aligned with CK's clustered key `(store, day, product, …)`.

- **Spend trends** is perfectly aligned to CK's clustered key (store, day, product, …). The plan is a short, sequential range scan with aggregation by product; CK is slightly faster for that reason.
- **Demographic, growth, and campaign** scan and join large portions of the fact table and then aggregate. SK's secondary indexes help a bit, but most of the time is spent in grouping and sorting, which limits the gap.[3]

---

5. Results: operational checks and explanations

10 Benchmark Scenarios (Notebook "parameters") ¶

#	Scenario / What we measured	Query/Feature exercised	Metric(s) captured	Winner (this run)
1	**Spend trends** (store + 30-day window → revenue by product)	Range scan on `(store, day)` + `GROUP BY product`	ms, rowcount	**CK** (5.47 vs 6.63 ms)
2	**Demographic influence** (income × age → revenue)	Scan fact + join `household_dim` + aggregate	ms, rowcount	**SK** (488 vs 564 ms)
3	**Growth categories** (30-day vs prior 28-day)	Two windowed scans + join `product_dim`	ms, rowcount	**SK** (1248 vs 1561 ms)
4	**Campaign impact** (coupon redeemed vs not)	Join fact ↔ redemption + aggregate	ms, rowcount	**SK** (6119 vs 6744 ms)
5	**Household lookup** (`SUM(qty)` by `household_sk`)	CK: no index → scan; SK: `idx_household`	ms, rows examined	**SK** (~7.8 ms vs ~1444 ms)
6	**Store + day micro** (same window as #1)	Pure index range scan timing	ms (top & range node)	**SK** (~26 ms vs ~43 ms)
7	**Insert throughput** (~1.42M rows)	Bulk insert into each design	ms, rows inserted	**CK** (23,434 vs 37,729 ms)
8	**Delete window** (store 140, days 6..36)	Localized range delete	ms, rows deleted	**CK** (91 vs 119 ms)
9	**Storage footprint**	`information_schema.tables`	data_mb, index_mb, total_mb	**CK** (98.7 MB vs 184.8 MB)
10	**Maintenance / OPTIMIZE**	Table rebuild + analyze	seconds	**CK** (25 s vs 53 s)

(Bonus check we also ran: buffer-pool impact — CK created ~26k extra read requests and ~12,490 disk reads vs SK's ~8k and ~89 for the household lookup; SK is far more cache-friendly for that access pattern.)

- ### Household-centric lookup

  CK has no index on household_sk alone, so it scans ~1.42M rows. SK uses idx_household and jumps straight to ~1,170 matches. In our run, CK was ~**1,444 ms** vs. SK ~**7–8 ms**. Buffer-pool deltas told the same story: CK added ~**+26k** logical requests and **+12,490** disk reads, while SK added ~**+8,109** and **+89**, respectively. This is exactly what MySQL documents: sequential scans trigger read-ahead and touch more pages; targeted secondary-index probes do not.[2],[4]

- ### Store + day micro timing (same window as spend trends).

  Both designs executed tight index **range scans** and finished in a few milliseconds. Plans looked as expected; the differences were trivial and consistent with the number of bytes each path touches.[3]

- ### Insert throughput (~1.42M rows).

  CK **23,433 ms**; SK **37,729 ms**. SK pays to maintain two secondary indexes per row; CK updates only the clustered key.[2]

- ### Delete window (store 140, days 6..36; ~208 rows).

  CK **91 ms**, SK **118.51 ms**. In CK, the day range forms a contiguous slice in the clustered index; SK must also update its secondaries.

- ### Storage footprint.

  bench_lines_ck ≈ **98.7 MB** total; bench_lines_sk ≈ **184.8 MB** total. The difference is index bytes, which also influence buffer-pool residency.[2]

- ### Maintenance (OPTIMIZE TABLE).

  CK **25 s**; SK **53 s**. InnoDB implements OPTIMIZE as a rebuild + analyze; more and larger indexes mean longer maintenance windows.[5]

---

6. Discussion

Why CK won "spend trends."

The predicate matches the left prefix of the clustered key. In InnoDB, that means short, sequential leaf scans and minimal random I/O.[1],[2]

Why SK often wins on the other deliverables.

When the workload is scan + join + aggregate, the benefit of physical clustering shrinks. Secondary indexes can reduce bytes read or become covering for a query (if columns are included), though larger indexes trade off insert and maintenance cost.[2],[6]

What the buffer-pool counters add.

Counters confirmed the mechanisms above: read-ahead and more misses for table scans; far fewer for targeted index probes.[4]

---

7. Limitations

Timings move with cache warm-up; we mitigated by repeated runs and used representative values. We did not add experimental covering indexes beyond the two SK secondaries due to user privileges. Single-node Docker results are hardware-specific, but the **relative** patterns are robust.

---

8. Recommendations

- If the bulk of reporting is **aligned to (store, day)**, CK is attractive: faster loads, smaller tables, shorter rebuilds.

- If analysts frequently slice by **household** or other non-aligned dimensions, SK with a **minimal** set of secondaries (e.g., store,day,product and household) delivers better query latency.
- For the hottest reports, consider **covering indexes** or **summary tables**; budget for the index size and maintenance overhead.
- Schedule OPTIMIZE/ANALYZE in low-traffic windows; SK tables will take longer.[5]

---

9. Conclusion

SK won **three of four** business queries and dominated the household lookup. CK won the aligned spend-trends query and was clearly better on load time, table size, and maintenance. The right design depends on the workload mix. Many teams land on SK with a small, well-chosen set of secondary indexes, and add coverings or summaries only where the win is clear.

Notes

1. *MySQL 8.0 Reference Manual*, "EXPLAIN Statement" (and TREE/ANALYZE output).
2. *MySQL 8.0 Reference Manual*, "Clustered and Secondary Indexes" (InnoDB) and related indexing chapters.
3. *MySQL 8.0 Reference Manual*, "EXPLAIN ANALYZE"—operator timing and plan verification.
4. *MySQL 8.0 Reference Manual*, "InnoDB Buffer Pool" and "Read-Ahead" (prefetch) behavior.
5. *MySQL 8.0 Reference Manual*, "OPTIMIZE TABLE"—InnoDB rebuild and analyze semantics.
6. (For background) PlanetScale, "Covering Indexes—MySQL for Developers," March 9, 2023.

Bibliography

- Oracle. *MySQL 8.0 Reference Manual*. Sections on EXPLAIN, InnoDB clustered and secondary indexes, InnoDB buffer pool and read-ahead, and OPTIMIZE TABLE.
- PlanetScale. "Covering Indexes—MySQL for Developers." March 9, 2023.