ALT/FNDATA ACADEMY · STUDENT STUDY SHEET

Study sheet: Module 2, data science and SQL

This session is a hands-on SQL workshop built on a real institutional dataset rather than a toy database. Students practice filtering, grouping, and aggregating transaction-level records to build a simple demand index, one of the core analytical patterns used by working data analysts. The dataset (watches, jewelry, handbags, fine art, and more, drawn from over 100 auction houses with history to the late 1990s) is large and messy enough to be realistic, while the sandbox environment removes any setup friction, so class time goes entirely to query design and interpretation. Students who want a coding extension can also see the same query issued against the production API using a short Python snippet, but the graded exercise stays entirely in SQL.

What you should be able to do

1. Write SELECT statements with WHERE, GROUP BY, and ORDER BY clauses against a real multi-million-row dataset.
2. Construct an aggregate metric (a brand-level demand index) from raw transaction records using quarterly bucketing.
3. Compute a ratio metric (pricing power, defined as median realized price divided by high estimate) and explain what values above or below 1.0 mean.
4. Identify and handle a data quality issue, specifically unsold lots, without discarding rows that matter to the analysis.
5. Translate a SQL query result into a short written interpretation suitable for a non-technical stakeholder.
6. (Extension) Reproduce a sandbox query as a POST request against a REST API and compare the two workflows.

Key terms

- **Date bucketing:** grouping timestamped rows into fixed periods, such as quarters, using a function like DATE_TRUNC so trends can be read at a coarser grain than individual transactions.
- **Demand index:** an aggregate count and average price built by bucketing sold lots over time, used here as a simple proxy for brand-level activity.
- **Pricing power:** the ratio of median realized price to median pre-sale high estimate; a value above 1.0 means buyers paid over the auction house's forecast.
- **APPROX_PERCENTILE:** the SQL function used to compute a median, or any percentile, efficiently over a large result set, preferred here over AVG because it is less sensitive to a handful of extreme prices.
- **Aggregate function:** a SQL function such as COUNT, AVG, or APPROX_PERCENTILE that collapses many rows into a single summary value per group.
- **Null guard:** a WHERE condition, such as sale_estimates_high_usd_price > 0, added specifically to keep a division from failing or producing a misleading result on missing or zero values.
- **Pagination:** the practice of retrieving a large result set in pages using limit and offset, relevant once a query run through the production API returns more rows than a single request should carry.

Core sandbox query

The query the module is built around. Run it, then change the brand or category and compare.

```
SELECT designer, item_title, sale_date, usd_price_decimal
FROM all_jewels_gems_data
WHERE designer LIKE '%Van Cleef%'
  AND status = 'sold'
ORDER BY sale_date DESC
LIMIT 50;
```

Common mistakes to avoid

1. Computing pricing power without a guard on the denominator. If sale_estimates_high_usd_price is null or zero for any row, the ratio can return null or an unreliable number; always add sale_estimates_high_usd_price > 0 to the WHERE clause before dividing.
2. Using AVG(usd_price_decimal) when a median would tell a cleaner story. A single exceptional lot can pull an average far higher than what most buyers actually paid, while APPROX_PERCENTILE is far less affected by that one row.
3. Grouping by quarter without also grouping by designer when a query covers more than one brand, which blends two brands' activity into a single misleading quarter-by-quarter line.
4. Treating the most recent one or two quarters in a demand index as a rising or falling trend. Recent periods are still having records added to them, so a dip or spike at the very end of the series usually reflects ingestion timing, not the market.
5. Misspelling or under-specifying a designer filter, forgetting the % wildcards or using an unusual abbreviation, and quietly working with a much smaller result set than intended without noticing the row count is low.
6. Assuming the sandbox SQL editor and the production API filter operators are identical without checking. The API's filters array uses explicit op values like "eq" and "like", which map to SQL logic but are not written the same way.

Full lesson, more queries, and the homework are in Module 2 at academy.altfndata.com. Questions: info@altfndata.com.