TCGBERG · ANALYTICS METHODOLOGY

Market Cap Methodology

The aggregation layer between fair value and indices: how we compute the total value of every printing-grade combination.

Version 1.0 | May 2026
Status: Specification — ready for implementation
Author: CK
Scope: per-printing, per-grader, per-grade, per-date
Depends on: SCRUM-61 (Fair Value Engine)

PROBLEM STATEMENT

1. Why Market Cap Matters

Fair value tells us what one specific copy of a card is worth. But that's only half the story — to understand a card's total economic significance we also need to know how many copies exist at that grade level. The product of fair value and population is the market cap: the total value of all known graded copies of a specific card variant at a specific grade.

Market cap is the foundational atomic unit for the platform's analytics layer. Indices need it to weight constituents. Deal screeners need it to identify whether a listing represents a meaningful share of total supply. Aggregations need it to answer questions like 'what is Charizard 1st Edition worth in total?' or 'what fraction of Pokémon Base Set value sits in PSA 10s?'

Without a market cap layer, every downstream consumer would compute it independently — leading to inconsistent definitions and duplicated logic. The market cap engine centralizes the calculation in one canonical place.

DEFINITIONS

2. The Atomic Unit

The atom is one (printing, grader, grade) tuple on a specific as-of-date. Examples of atoms: 'Charizard 1st Edition Shadowless graded by PSA at grade 9 on May 10, 2026.' 'Venusaur Unlimited graded by CGC at grade 8 on March 14, 2026.' Each atom resolves to a single market cap value.

```
atom_market_cap = atom_fair_value × atom_population
```

Both inputs are sourced from existing canonical tables. `fair_value` comes from the Fair Value Engine (SCRUM-61), reading `fair_values.value`. `population` comes from `graded_population.population`, which is populated by the existing population ingestion pipeline (PSA Population Reports).

All values are USD-denominated, matching the fair value engine's output convention. Currency conversion does not happen at the market cap layer.

ARCHITECTURE

3. Atoms First, Aggregates Computed at Query Time

The market cap engine writes only atomic rows. Aggregations across higher dimensions (by card, by set, by grader, by grade tier, etc.) are computed via SQL `GROUP BY` queries at the point of consumption rather than pre-stored.

Reasoning:

- **Postgres handles this trivially.** Pokémon Base Set produces ~1.7 million market cap rows over a year of daily history. Aggregations via SUM/GROUP BY on indexed columns return in milliseconds.

- **Query patterns are unknown until users surface them.** Pre-aggregating without that knowledge optimizes the wrong things and creates refresh-coordination overhead.
- **Indices consume atoms directly.** An index's constituents are at the (printing, grader, grade) level — atoms are the natural input.
- **Easy to add later.** If specific aggregations become bottlenecks, materialized views or aggregation tables can be added incrementally without changing the atom layer.

This decision is reversible. If query performance degrades on real workloads, the atom layer remains unchanged while caching layers are introduced above.

SCHEMA

4. The printing_market_caps Table

Column	Type	Purpose
printing_id, grader_id, grade_id, as_of_date	FK + Date	Primary key (unique together)
fair_value	Numeric(18,2)	Cached input from fair_values.value (USD)
population	Integer	Cached input from graded_population.population
fair_value_confidence_score	SmallInt	Cached from fair_values.confidence_score
market_cap	Numeric(20,2)	fair_value × population (USD)
currency	Varchar(3)	Always 'USD' for now
fair_value_as_of_date	Date	The as_of_date of the fair_value used
population_as_of_date	Date	The as_of_date of the population used
created_at, updated_at	Timestamp	Audit

Indexes:

- `(as_of_date DESC)` — for daily scans
- `(printing_id, as_of_date DESC)` — for time-series per printing
- `(as_of_date, market_cap DESC)` — for top-N queries
- Unique on `(printing_id, grader_id, grade_id, as_of_date)`

Why cache fair_value, population, and confidence?

Each market cap row carries the inputs that produced it. This makes audit queries cheap — answering 'why is this market cap weird?' requires no JOINs. The storage cost is approximately 30 bytes per row, paid back many times over in query simplicity and dashboard responsiveness.

ALGORITHM

5. Computing One Atom

For each (printing, grader, grade, as_of_date) tuple:

- Look up the most recent `fair_values` row for this tuple where `as_of_date <= target_date`. If none exists, skip.
- Look up the most recent `graded_population` row for this tuple where `as_of_date <= target_date`. If none exists, skip.
- Compute `market_cap = fair_value × population`.
- Upsert one row into `printing_market_caps` with the cached inputs and the result.

The 'most recent on or before target_date' lookup pattern matters. Population data is published monthly by PSA, while fair value updates daily. On any given day, the most recent population reading might be 0–30 days old, which is acceptable. The market cap reflects today's fair value applied to the latest known supply.

Both `fair_value_as_of_date` and `population_as_of_date` are persisted in the row, so consumers can see exactly how stale each input was at compute time.

EDGE CASES

6. Missing Data Handling

Condition	Behavior	Reasoning
Fair value missing	Skip — no row written	Cannot compute price × supply without price
Population missing	Skip — no row written	Cannot compute price × supply without supply
Both missing	Skip — no row written	Nothing to compute
Population = 0	Skip — no row written	Zero-pop combos are noise
Fair value confidence very low	Compute; consumers filter on confidence	Don't gate at this layer; let downstream decide
Fair value is NULL	Skip — no row written	Same as missing
Stale population (30+ days old)	Compute normally	Population is monthly cadence; stale is normal

> **Design principle: never invent data.** The market cap table only contains fully-defined values from real inputs. If either input is missing, no row is written. This means every row in the table represents a genuine market cap, not an approximation. Downstream consumers can always trust the data they see.

CONSUMERS

7. Aggregation Helpers

The atom layer exposes a small set of canonical aggregation helpers in `app/analytics/market_cap.py`. These functions wrap common aggregation queries so consumers don't reinvent them.

Function	Returns	Use Case
market_cap_by_printing(printing_id, as_of_date)	Decimal	Sum across all (grader, grade) for one printing
market_cap_by_card(card_id, as_of_date)	Decimal	Sum across all printings of a card
market_cap_by_set(set_id, as_of_date)	Decimal	Sum across all cards in a set
market_cap_by_grader(grader_id, as_of_date)	Decimal	Total measured by one grader
market_cap_by_set_grader(set_id, grader_id, as_of_date)	Decimal	Set value as measured by one grader
top_market_caps(n, as_of_date, level)	List	Top N at the printing/card/set level
market_cap_share(printing_id, set_id, as_of_date)	Decimal	What % of set value this printing represents

All helpers are implemented as straightforward parameterized SQL via SQLAlchemy's `text()` with named bind parameters, matching the codebase's existing pattern. Performance is bounded by index lookup speed.

INTEGRATION

8. Downstream Consumers

Consumer	How It Uses Market Cap
Indices Engine	Constituent weights via market_cap_by_printing or atom level; cap rules applied
Card Pages	Show 'total value of all PSA 9 copies' alongside fair value
Set Pages	Total set value, breakdown by grade and grader
Top Movers	Rank cards by absolute market cap change over time windows
Deal Screener	Use market cap to surface 'meaningful' deals (high market cap + listing < fair value)
AI Commentary	Newsletter: top movers by market cap dollar change

EXAMPLES

9. Worked Examples

Example 1 — Charizard 1st Edition Shadowless PSA 9

Inputs:
fair_value = $13,000.00 (confidence 91)
population = 1,200
Output:
market_cap = $15,600,000.00
Interpretation:
The total value of all known PSA-9-graded Charizard 1st Edition Shadowless cards in existence is approximately $15.6 million.

Example 2 — Aggregating Charizard Across All Variants and Grades

Query: `market_cap_by_card(charizard_id, today)`
Implementation: SUM over all (printing, grader, grade) atoms where card_id matches.

```
SELECT SUM(market_cap)
FROM printing_market_caps mc
JOIN printings p ON p.id = mc.printing_id
WHERE p.card_id = :card_id
  AND mc.as_of_date = :as_of_date
```

Returns the aggregate market cap of every Charizard variant at every grade level. This is the headline 'Charizard total value' number that would appear on a card page or be reported in a newsletter.

Example 3 — Set Composition by Grade Tier

Question: 'What fraction of Pokémon Base Set's PSA-graded value sits at each grade level?'
Answer: GROUP BY grade.numeric_value, SUM(market_cap), express as % of total.

Typical result for vintage sets: PSA 10 dominates (40-60% of value), PSA 9 next (20-30%), PSA 8 and below tail off. Reflects the steep premium scaling at higher grades.

Example 4 — Edge Case: Card With Sales But No Population Data

Inputs:
fair_value = $450 (confidence 73)
population = NULL (PSA hasn't published populations for this printing yet)
Behavior:
No row written. The market cap engine logs an info-level skip and continues. When PSA publishes population data, the next daily run will write the row.

OPERATIONS

10. Daily Run and Performance

Market cap computation is daily, run after fair value computation completes:

- Iterate every (printing, grader, grade) tuple where both a fair value AND a population row exist for as_of_date.
- For each, look up the most recent fair value and population, multiply, upsert.
- Idempotent — re-running for the same as_of_date produces identical results.
- Job report (success count, skip count, duration, errors) written to `job_runs`.

Expected runtime for Pokémon Base Set (~1,082 atoms per day): under 30 seconds. Bulk INSERT ... ON CONFLICT pattern with the same Supabase pooler workaround (`SET LOCAL statement_timeout = 0`) used in the fair value engine.

Backfill is supported via `--start-date` and `--end-date` CLI args. Backfilling 6 months × 1,082 atoms produces ~196,000 rows in approximately 10 minutes (much faster than the fair value backfill since each computation is a single multiplication rather than a 9-stage algorithm).

INSPECTION

11. Inspector Dashboard

A new page `frontend/price-dashboard/market-caps.html` extends the existing dashboard, mirroring the fair value inspector pattern. Five sections:

Section	What It Shows
Filters (sidebar)	Set, card, printing, grader, grade, as-of-date — same cascading pattern as fair-values
Headline panel	Atom market cap, fair_value × population breakdown, confidence badge, change percentages
Composition treemap	How total value of selected scope splits across grade levels
Time series	Market cap over the last N days; secondary line for population if it changes
Top constituents	Top 20 atoms by market cap for the selected scope (set or card)

Five new endpoints in `scripts/price_dashboard.py`, matching the existing pattern (raw SQL via `_fetch_all`, named bind params, hand-rolled HTTP routing).

LIMITATIONS

12. Known Limitations

- Population staleness is invisible to the market cap value. A 25-day-old population reading produces the same multiplication as a fresh one. The `population_as_of_date` column makes this auditable but doesn't change behavior.
- Cross-grader aggregations sum population from PSA + BGS + CGC. If the same physical card has been crossed (regraded by another company), it would be double-counted. This is a known data quality issue not addressable at the market cap layer.
- Raw (ungraded) cards are not included. The `graded_population` table only tracks graded supply. A separate raw_market_cap layer could be added later if needed.
- Pre-grading population (cards in the wild that haven't been graded) is invisible. Market cap reflects only the documented graded supply, not the latent total.
- Confidence is cached from fair value but not propagated as a separate market cap confidence. Consumers filter on `fair_value_confidence_score >= N` to require trustworthy inputs.

This methodology document is the spec for ticket SCRUM-62 in the TCGBerg backlog. Implementation details (table DDL, module structure, test coverage) are captured in the ticket itself.

Companion documents: Fair Value Methodology (SCRUM-61), Index Methodology, Pokémon Data Sources.
Document version 1.0 — May 2026.