

Dremel Paper.pdf
PDF

I am a high agency individual who uses first principle thinking to understand the value of things built in this world and how does it affect and patch up in the current world

I am attaching this paper of Dremel (<https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/36632.pdf>) and you are supposed to explain me that - NOT JUST IN DETAIL - but by explaining why does it matter - no generic explanations - REASON RECURSIVELY AND FIGURE OUT WHY IT IS A BIG DEAL? - Then explain me HOW DOES IT SOLVE THE PROBLEM THAT WAS A BIG DEAL - Then explain the most nitty gritties of the approach taken - IN the end explain different approaches taken to solve that same problem in past, and now

DO NOT FUCKING WASTE MY TIME - EVERYTHING THAT YOU SAY SHOULD BE EXTRAORDINARILY VALUABLE - NO BULLSHIT

Thought for 1m 52s >



Dremel — why this paper actually mattered

1. It changed the time-constant of thinking with data.

At Google scale, analysts needed answers in seconds, not hours—on petabytes of nested (logs, events, documents) stored in place (GFS/Bigtable). Dremel showed you can run SQL-like aggregations over **trillion-row tables in seconds**, directly over shared storage, without loading into a DBMS or staging MapReduce pipelines. That shift—from “batch jobs” to “interactive loops”—is the real step change: it compresses

the hypothesis→answer→next→... which is what creates product velocity and operational insight. [Skip to content](#) [Google Research](#)

2. It made “SQL on semi-structured data at web scale” practical.

Before Dremel you either: (a) normalized/flattened nested data (expensive and lossy at scale), or (b) wrote multi-stage MR jobs (hours). Dremel couples a **columnar layout for nested records** with a **multi-level execution/serving tree**, so you read far less, parallelize far more, and tolerate failures/stragglers—yielding the interactivity that MR and many parallel DBs couldn’t reach at thousands of nodes. [Google Research](#)

3. It seeded an ecosystem.

Its nested-columnar encoding (definition/repetition levels) became the blueprint for Parquet and BigQuery’s RECORD/REPEATED types. That is why essentially every modern lakehouse engine speaks “Dremel” when dealing with JSON-like schemas at speed. [Google Research](#) +2

How Dremel solved the hard parts

1. Store nested data as columns (not rows) without losing structure.

Dremel introduces **repetition** and **definition** levels per value.

- *Repetition level*: where a repeated field in the path has repeated (disambiguates siblings vs new parent instances).
- *Definition level*: how many optional/repeated ancestors are actually present (so NULLs and parent presence are reconstructable).

Levels are bit-packed (only as many bits as needed), NULLs are implied by levels, and levels can often be elided for always-defined fields. Result: you can scan only the columns you need, even for deeply nested schemas. [Google Research](#)

2. Assemble only when you must; otherwise, avoid it.

When full records are needed, Dremel builds a small finite-state machine to **reconstruct rows** from columns using repetition levels as transitions—linear-time, streaming assembly. But for aggregates (the common case), Dremel **bypasses assembly** and computes directly over column streams annotated with levels. This is ridiculously efficient. [Google Research](#)

3. Execute through a multi-level serving tree.

A root rewrites the query into per-tablet subqueries, pushes them down the tree; leaves scan local/nearby tablet stripes; **intermediate servers aggregate on the way up**. This makes “many-group” aggregations scale (root avoids serially merging thousands of partials). The paper shows speedups simply by adding an intermediate level for a wide GROUP BY. [Google Research](#)

4. Engineer for tail tolerance and concurrency.

- **Prefetch** column blocks asynch'ead cache ≈95% hit rates. Skip to content
 - **Tablet replication**: fail over to another .. one is slow/unavailable.
 - **Partial-scan thresholds** (e.g., return after 98% tablets) to cut straggler tails.
 - **Query dispatcher** schedules by priority and manages slots (leaf threads).
- These are pragmatic choices that turn theory into predictable 99th-percentile latency.

Google Research

5. Prove it at scale.

Experiments compare MR on records vs MR on columns vs Dremel: MR→hours, MR+columnar→minutes, **Dremel→seconds** on an 85-billion-record table (3000 nodes). They also include tables with **1+ trillion records** (multi-TB, hundreds of thousands of tablets).

Google Research

The nitty-gritty (what actually makes it fast and robust)

1. Nested-column encoding details

- Each **column** = blocks of compressed values + packed (r,d) level streams.
- Levels use as few bits as the schema depth demands; NULL is inferred by $d <$ max definable depth for the path.
- Base algorithms compute levels even for missing fields efficiently (sparse schemas with thousands of fields, only ~100 present per record are common).

Google Research

2. Record shredding & assembly FSM

- **Shredding**: a tree of “field writers” mirrors the schema; children inherit parent level state and only sync on new values—avoids pushing parent state down the tree unnecessarily.
- **Assembly**: FSM state per selected field; transitions labeled by repetition level choose the next leaf to read; for aggregates, Dremel emits results with correct (r,d) *without assembling*.

Google Research

3. Execution plan inside a server

- Vectorized/iterator plan scans selected columns **in lockstep**, evaluates expressions, and pushes **partial aggregates** upward. Type-specialized code for scalar functions + one-pass approximations (top-k, count-distinct) keep it single-scan.

Google Research

4. Serving tree topology trade-offs

- 2-level (root→leaves) can bottleneck the root on wide groupings.
- 3-level (root→100 mids→2900 leaves) halves latency for group-heavy queries, while small-group aggregations saturate at 3 levels. This is a principled **fan-out/fan-in** design to match aggregation cardinality.

Google Research

5. Storage in place, no slow “load”

- Operates **in situ** over GFS/BigTable. Skip to content → so you skip multi-hour load phases typical of DBMSs—critical when ~~the database is~~ “run dozens of MR analyses before the DB finishes its first load.” Google Research

What came before, and what came after

Before Dremel (circa 2006–2010)

1. MapReduce, Pig, Hive

You could express analysis over nested logs, but you paid in latency (minutes→hours) and operational friction (stages, joins, temp data). Hive/Pig translate to MR; Dremel executes natively and interactively. Google Research

2. Parallel DBMS / MPP row-stores & column-stores

MPP databases brought SQL and parallelism but rarely scaled to **thousands** of nodes in published reports, and didn’t natively solve *nested* data over shared storage. Dremel explicitly targets that scale and model. Google Research

3. XML stores

They separated structure/content but struggled with flexible models and selective column retrieval; not designed for interactive scans over petabytes. Dremel’s column-striped nested layout is different in purpose and mechanics. Google Research

After Dremel (2011 → now)

1. BigQuery (commercialized Dremel)

Dremel’s **serverless, pay-per-use** model became BigQuery: no provisioning, elastic parallelism, nested/repeated columns, seconds-level analytics at petabyte scale. This is the mainstreaming of Dremel’s ideas. Google Research +1

2. Parquet & ORC (the file formats)

They **adopt Dremel’s repetition/definition encoding** for nested columns. That’s why Spark/Presto/Trino/ClickHouse/DuckDB can query semi-structured data efficiently today—everyone converged on this representation. Apache Parquet

3. Open-source engines

- **Apache Drill:** an OSS attempt to replicate Dremel’s interactive SQL on nested data.
- **Presto/Trino, Spark SQL, ClickHouse, DuckDB:** different execution models, but the same core lesson—columnar I/O, vectorized operators, pushdown, and lake-native formats—trace back to Dremel’s proof that interactivity at scale is achievable.

4. Variations within Google

Google later built **PowerDrill** for ultra-low latency on a few hot datasets (heavier in-memory/partitioning), trading breadth for speed; Dremel remained the general interactive workhorse. (Historical context; different design point.) WIRED

The shortest possible 'tel'

- **Data:** nested logs/docs in GFS/Bigtable.
- **Layout:** one column per leaf field + tiny level streams (r,d) to remember hierarchy.
- **Scan:** read only the needed leaf columns; compress well; skip NULLs cheaply.
- **Compute:** vectorized aggregation over columns; avoid row assembly; one pass.
- **Distribute:** rewrite the query across a **serving tree**; aggregate partials on the way up; tolerate tail via replication, prefetch, and 98%-done thresholds.
- **Result: seconds**, not hours, on **trillion-row** tables. Google Research +1

If you apply first-principles: shrink bytes read, minimize CPU per value, maximize parallel fan-out/fan-in, and engineer away the tail. Dremel is the first coherent system that did all four for **nested** data at Google scale—and that's why it's a big deal.



Based on all of this -> DRAFT QUESTIONS WHICH MATCH THE DIFFICULTY of questions attached below which are form sample quizzes --> ALSO add a few question that are TOUGHER than the questions in samples

The questions should not be absolute simple giveaways - but should make us think - I have attached few questions ON DIFFERENT PAPERS, which would help you understand the level of UNDERSTANDING REQUIRED TO SOLVE THESE QUESTIONS from sample so that you get an understanding of what is the level set

The questions you draft will help me study for my midterm and score marks to get a job and help my family - make sure that I get good marks

REASON RECURSIVELY and draft 5 questions

<sample>
Skip to content

1. How does DynamoDB use data partitioning to handle scaling?
2. What is the key problem that Dremel handles? How is it different or compared to BigQuery?
3. Borg employs a priority and quota system for job scheduling. Describe how this system works and explain how it contributes to balancing resource allocation across production and non-production tasks.
4. If the master node in GFS faces a sudden increase in workload due to high-frequency read requests, what steps could be taken to prevent bottlenecks?
5. Why is hybrid cloud beneficial for an enterprise compared to either public or private cloud?
6. What is paravirtualization and why is paravirtualization more efficient than full virtualization?
7. Explain how iterative memory copy is leveraged for live migration?
8. What are the key differences between VM and containers

</sample_questions>

these are questions on different papers like dremel, bigquery, borg, etc. --> but these are attached only for you to UNDERSTAND THE DIFFICULTY of questions that are asked on research papers

<task>

Based on this difficulty level you are supposed to DESIGN 5 QUESTIONS SOLELY BASED ON Dremel paper

</task>

Thought for 1m 32s >

Got it. Here are 5 high-yield, thinky questions on **Dremel** only—two marked **[HARDER]** to push beyond the sample level.

1. Columnar encoding for nested /definition levels). [HARDER]

You're given this Protocol Buffer and two records:

nginx

 Copy code

```
message Session {
    optional string user_id;
    repeated Group groups {
        required string country;
        repeated int64 ad_impressions;
    }
}
```

R1:

- user_id = "u1"
- groups = [
 { country = "US", ad_impressions = [2, 5] },
 { country = "FR", ad_impressions = [] }
]

R2:

- user_id = "u2"
- groups = []

Tasks:

- a) For column path groups.country , list the (value, repetition level, definition level) triples emitted across R1 and R2.
- b) For column path groups.ad_impressions , do the same. Include the necessary **nulls** implied by empty repeats.
- c) Using your DL/RL streams for part (b), show how COUNT(ad_impressions) and AVG(ad_impressions) can be computed **without reconstructing rows**, and explain exactly which DL thresholds you use for “value present.”
- d) Explain how these encodings preserve **in-record correlation** between country and its corresponding ad_impressions without materializing a Cartesian product.

2. Scan math: why columnar + compression wins for ad-hoc analytics.

A Dremel table has 10^{12} rows, 100 fields, average logical row size ≈ 1 KB uncompressed. A query touches only 3 columns: two 64-bit ints and one low-cardinality string used in a filter. Assume:

- Columnar compression: ints 10x, the string 20x (via dictionary),
- Row-store compression alternative: 3x over the whole row,

- Cluster can sustain 120 GB/s across all nodes. Skip to content

Tasks:

- Estimate total bytes read for Dremel vs a compressed row-store for this query, ignoring filter selectivity.
- If the predicate filters to 5% of rows and is applied during column scan, recompute Dremel bytes read (account for reading the string column fully but projecting only matching row ids for the two int columns).
- Translate your bytes into a **lower-bound latency** at 120 GB/s.
- From your numbers, articulate the **first-principles reason** Dremel achieves interactive latencies where MapReduce or a row-store cannot, even holding compute constant.

3. Serving tree + tail tolerance under failures/stragglers. [HARDER]

Dremel executes via a multi-level aggregation tree (root → intermediates → leaves).

You have 1,024 leaves; each leaf processes “shards” of ~256 MB compressed column chunks at ~160 MB/s median. Observed tail: 1% of shards run 5× slower; 2% of leaves can fail mid-query.

Tasks:

- Propose a **shard sizing and scheduling** policy (e.g., number of in-flight shards/leaf, speculative re-issuance threshold, work stealing) to keep p95 end-to-end query time under 3 s for a scan of 4 TB compressed data. Justify in terms of amortizing stragglers and hiding failures behind spare capacity.
- Precisely state which node failures **must** abort the query vs which can be masked by rescheduling, and why (think: leaf vs intermediate vs root semantics and statefulness).
- Explain why **stateless partial aggregates** and **associative/commutative** ops are critical for making (a) and (b) feasible without duplicate suppression overhead.

4. Correctness with repeated fields (avoiding bogus joins).

Consider a repeated nested record `visits` with fields (`country STRING`, `revenue FLOAT`) and this single logical row:

ini

 Copy code

```
visits.country = ["US", "FR", "US"]
visits.revenue = [ 10.0,  5.0,  2.0]
```

Tasks:

- Show the **wrong** result you’d get if you naïvely flatten `country` and `revenue` independently and then group by `country` (explain the overcount mechanism).
- Using Dremel’s nested column semantics (DL/RL), describe how a query engine can

aggregate SUM(revenue) BY country Skip to content **g the within-record alignment of positions.** Be explicit about which level to sum up to prevent cross-product explosion.

c) Generalize a rule of thumb: when querying repeated fields in Dremel, what constraints must a planner enforce to keep correlations intact, and how does the column encoding make that check cheap?

5. Design stress test: Dremel's read-most assumptions under write-heavy ingestion. [HARDER]

Dremel's wins come from read-only, compressed column chunks laid out for scans.

Now assume a new workload: continuous high-rate appends with late-arriving corrections, and analysts still demand sub-second ad-hoc scans.

Tasks:

- a) Propose an ingestion path that preserves most of Dremel's scan efficiency (think: immutable column chunks + small **delta columns**, periodic compaction, segment maps).
- b) Identify the **two biggest regressions** you expect (e.g., compression ratio, predicate pushdown locality, metadata amplification) and quantify qualitatively how they hurt latency.
- c) Specify what changes (if any) you'd make to the serving tree or query planner to keep tail latency in check when queries must stitch base + deltas at read time.
- d) Defend whether you'd still choose Dremel's architecture over a distributed row-store or a write-optimized columnar system for this mixed workload—and under what thresholds you'd switch.



For Every Question below write answers in following format:

<answer_format>

1. Explain in detail with good reasoning and in simple language
2. Use technical ALL THE **NECESSARY** TECHNICAL WORDS - clean, lean and crisp answers
3. 7 bullets

</answer_format>

||||||| Skip to content

questions:

1. Explain briefly but precisely how Dremel architecture is designed both based on concepts from Web Search and Parallel DBMS.
2. What are the two key reasons Dremel is so fast and why?

Thought for 28s >

1) Dremel architecture: how it borrows from Web Search + Parallel DBMS

- **Scatter-gather serving tree (Web Search DNA):** A root **query broker** fans out to hundreds/thousands of leaves via **intermediate aggregators**; leaves scan local shards; partials are **aggregated on the way up**—exactly the search-engine “**broker → shard → merge**” pattern for low latency at massive fan-out.
- **Shard replication + tail trimming (Web Search DNA):** Data is split into **tablets** (column stripes) with **replicas**; slow/failed leaves are **re-issued** to another replica (speculation) so the **p95/p99** latency stays tight—classic tail-latency control from web search.
- **Immutable segments over a distributed store (Web Search DNA):** Like **read-only index segments**, Dremel reads **immutable, compressed column blocks** directly from GFS/Bigtable; no locks, no writes, easy **prefetch/caching**, and predictable read performance.
- **Columnar storage + vectorized operators (Parallel DBMS DNA):** Queries touch only needed **columns**; operators are **vectorized** and **late-materialize** rows; **predicate pushdown** prunes data early—core column-store/MPP database techniques.
- **Two-phase (multi-level) aggregation (Parallel DBMS DNA):** Use **associative/commutative** partials at leaves, **merge** at intermediates, **finalize** at root; this is textbook **parallel GROUP BY** with in-network reduction.
- **SQL-on-nested with shredding/levels (Hybrid):** A **declarative SQL** dialect over **nested/REPEATED** fields via **definition/repetition levels**; this marries DBMS

- semantics (schemas, plans) with distributed execution.
- **Admission control + slot scheduling.** A central **dispatcher** controls **concurrency** (leaf **slots**, priorities), balancing throughput vs latency—combining web search's request scheduling with MPP query resource management.
-

2) Two key reasons Dremel is fast — and why they matter

- **Reason #1 — Columnar over nested data (I/O minimization):** Repetition/definition levels let Dremel store **nested** structures as **columns** without flattening; queries read **only needed columns**, skipping NULLs cheaply—dramatically fewer bytes off disk/network.
- **Compression that actually works (amplifies #1):** Column homogeneity enables dictionary/RLE/bit-packing; nested levels are tiny **bitstreams**; less data read ⇒ **lower latency** and more **cache** effectiveness.
- **Predicate & projection pushdown (amplifies #1):** Filters are applied as **columns stream**, cutting downstream work; **late materialization** avoids assembling rows unless strictly needed—**CPU and memory** stay low.
- **Direct compute on columns (amplifies #1):** Common aggregates (COUNT, SUM, AVG) operate directly on **column vectors** with level-aware logic—**no row reconstruction** in the hot path.
- **Reason #2 — Multi-level serving tree (parallelism + tail control):** A **root** → **intermediates** → **leaves** tree provides **massive parallelism** and **in-network aggregation**, shrinking return payloads and avoiding root bottlenecks.
- **Straggler masking & failover (amplifies #2): Speculative re-issues, replica reads, and partial-result thresholds** (e.g., return at 98% tablets) tame **p99 tails**—**interactive** even at web scale.
- **Zero-load architecture tax (supports both):** Operates **in situ** over GFS/Bigtable (read-only), so there's **no ETL/load** phase; interactive queries start **immediately** and scale elastically with available leaves.



NEXT QUESTIONS:

3. Discuss the specific challenges and trade-offs involved

in implementation "s nested columnar storage format
Skip to content compared to flat relational column stores.

4. What is the key problem that Dremel handles? How is it different or compared to BigQuery?

Thought for 46s >

3) Dremel's nested columnar vs flat relational column stores — challenges & trade-offs

- **Correctness invariants (RL/DL):** Dremel must encode hierarchy with **repetition/definition levels** so sibling lists stay aligned; planners must be **path-aware** (e.g., group by within a repeated scope) or you get **cartesian blow-ups** and wrong aggregates. Flat column stores are **positionally aligned 1:1**, so planners are simpler.
- **Shredding & assembly cost:** Ingestion must **shred** records into many leaf columns + level streams; some queries must **assemble** rows with a small FSM. Flat stores avoid level streams and rarely reconstruct rows; Dremel gains flexibility on nested data but pays **CPU/cache** for level handling.
- **Write path vs read path:** Dremel favors **immutable, append-only** column blocks and periodic **compaction**; random updates and transactional writes are expensive. Traditional column stores (and many row stores) offer **updatable segments** and **index maintenance**, trading some scan speed for update flexibility.
- **Compression variability:** Nested sparsity + dictionaries + RLE often compress **extremely well**, but **irregular repeats** (wide, jagged arrays) weaken run-length locality and can fragment blocks; flat schemas yield **more predictable compression** and simpler **null bitmaps**.
- **Vectorization & stream alignment:** Operators run on **column vectors**, but multiple leaf streams must stay synchronized via **repetition levels**; this constrains prefetch and can introduce **branching** in tight loops. Flat stores keep **lockstep positions**, maximizing straight-line SIMD/vector gains.
- **Statistics, indexing, pushdown:** Collecting **min/max, histograms, bloom filters** for **nested paths** is harder; selectivity estimation for deep fields is noisier, so **predicate pushdown** and **join planning** can be less precise than for flat columns with simple key distributions.
- **Interoperability & SQL semantics:** Dremel extends SQL with **RECORD/REPEATED** types; tools must understand nested scopes and **late materialization** rules. Many BI

tools assume **flat relational** tab¹ Skip to content ters, view-flattening, or training to avoid semantic traps.

4) What core problem Dremel solves, and how that differs from BigQuery

- **Core problem (Dremel):** Deliver **interactive SQL** over **massive, read-mostly, nested** datasets (logs/events/docs) stored **in situ**, with **seconds-level latency**—eliminating multi-hour ETL or MapReduce pipelines for ad-hoc exploration and ops analytics.
- **Dremel's technical answer:** A **nested columnar** layout (repetition/definition levels), **projection/predicate pushdown**, **vectorized level-aware operators**, and a **multi-level serving tree** (root→intermediate→leaves) with **in-network aggregation** and **straggler masking**.
- **How BigQuery differs (productization):** BigQuery is Dremel's architecture as a **serverless, multi-tenant managed service**—no cluster ops, **on-demand elasticity**, isolation, and SRE-grade reliability layered atop the core execution ideas.
- **Storage & system model:** Dremel (paper) queries **read-only column blocks** in Google's internal storage; BigQuery adds **managed columnar storage**, **separation of storage/compute**, **automatic replication**, and **governance** (IAM, audit).
- **SQL & features:** Dremel's paper shows a focused SQL over nested data; BigQuery offers **Standard SQL**, **UDFs**, **window functions**, **partitioning/clustering**, **time-travel**, **materialized views**, and **fine-grained ACLs**—broadening workloads beyond exploratory scans.
- **Data ingestion & federation:** Dremel assumes batch, immutable segments; BigQuery supports **streaming inserts**, **batch loads**, and **federated queries** (e.g., external objects/lakes), plus **automatic schema evolution** workflows.
- **Economics & workload management:** Dremel is an internal system; BigQuery adds **bytes-scanned pricing**, **slot reservations/autoscaling**, **workload management**, and **caching**—turning the same architectural core into a predictable, **pay-for-what-you-use** analytics platform.



Give both answers in a more detailed way for every bullet

3) Dremel's nested columnar vs flat relational column stores — challenges & trade-offs (detailed, simple, high-impact)

1. Keeping hierarchy correct with RL/DL (correctness over speed):

- Dremel adds **Repetition Level (RL)** and **Definition Level (DL)** to every column value so the engine knows which list item belongs to which parent record.
- Why it's hard: a flat column store assumes "row 7, col A" aligns with "row 7, col B." In nested data, **positions don't line up** unless you track them.
- Impact: extra metadata to compute, store, and read; but you **avoid wrong joins and double counting** when aggregating inside repeated fields.

2. Shredding on write, assembly on read (extra CPU paths):

- **Shredding** splits one nested record into many leaf columns + tiny RL/DL bitstreams; **assembly** re-stitches only when full rows are required.
- Trade-off: flat stores skip this complexity but struggle with JSON-like data; Dremel pays some **ingest CPU** and an occasional **reconstruction cost** to get **fast, selective scans**.

3. Read-optimized, write-constrained (immutability bias):

- Dremel favors **immutable, append-only column chunks** with periodic compaction; random updates are expensive.
- Flat/row stores often support **in-place updates** and indexes more easily.
- Impact: for logs/analytics (append-heavy, read-mostly), Dremel flies; for OLTP-style updates, you'd pick a different system or add **delta layers**.

4. Compression: excellent on regular nests, weaker on jagged lists (data-shape sensitive):

- Columnar + dictionaries/RLE **crush bytes** when fields repeat or are sparse.
- But **irregular arrays** (e.g., wildly varying list lengths) break runs and reduce locality, so compression and scan efficiency **drop**.
- Flat schemas compress more **predictably** but lose nested fidelity.

5. Vectorization with alignment logic (hot loops with branches):

- Dremel's operators are **vectorized**, but they must step through **multiple level streams** in lockstep to keep siblings aligned.
- This introduces **branching** and pointer chasing that flat stores largely avoid.

- Impact: still fast, but SIMD is lower than perfect flat scans; careful engineering reduces ... Skip to content

6. Statistics and pushdown on paths, not just columns (harder planning):

- Optimizers need **min/max, NDV, histograms** for paths like `orders.items.price`, not just `price`.
- Estimating selectivity on deep fields is **noisier**, so predicate pushdown and join choices can be **less exact** than in flat tables.
- Impact: you sometimes scan a bit more than ideal; engines compensate with **cheap skipping** and **wide parallelism**.

7. Tooling and SQL semantics (learning curve + adapters):

- SQL over **RECORD/REPEATED** types adds rules (grouping by repeated scopes, avoiding cross-products).
- Many BI tools expect **flat tables**; you may need **views, UNNEST** helpers, or training.
- Impact: once teams learn these patterns, they **retain structure**, cut scans, and avoid data loss from forced flattening.

4) The core problem Dremel solves, and how that differs from BigQuery (detailed, simple, high-impact)

1. The problem in one line (why it exists):

- Analysts had **huge, nested, append-only data** (logs, events, telemetry) and needed **SQL answers in seconds**, not hours.
- Traditional paths (ETL to DB or MapReduce) were **too slow** for interactive thinking.

2. Dremel's core idea (how it wins):

- Store nested data **as columns** with **RL/DL levels**, scan **only needed columns**, and compute **directly on columns** (no row rebuild unless required).
- Result: **far fewer bytes read + vectorized math** = interactive latency on **trillion-row tables**.

3. Dremel's scale engine (serving tree):

- A root → **intermediates** → leaves tree **fans out** work and **aggregates in-network** on the way back.
- Stragglers are masked with **replica re-reads** and **early completion** (e.g., 98% done), keeping **p95/p99** tight.

4. What BigQuery changes (product vs paper):

- BigQuery = Dremel + cloud productization.**

- It's **serverless** (no cluster to manage), **multi-tenant**, with **automatic scaling**, **security/IAM**, **billing by bytes scanned**, and **SLA-grade reliability**.

5. Storage/compute model differences (operational reality):

- Paper Dremel runs on **internal Google storage** over **read-only column chunks**.
- BigQuery adds **managed columnar storage**, **separation of storage and compute**, **partitioning/clustering**, and **automatic replication**, so teams just **load and query**.

6. SQL & features (breadth of workloads):

- Dremel's paper shows a focused SQL over nested data for analytics.
- BigQuery ships **Standard SQL**, **UDFs**, **window functions**, **materialized views**, **time travel**, **federation** (e.g., external tables), and **fine-grained governance**—supporting **ad-hoc**, **dashboards**, and **pipelines**.

7. Economics & ops (why orgs adopt BigQuery):

- Dremel proves the **engineering**; BigQuery makes it **frictionless: pay for bytes**, **autoscale**, **no ops team**, **workload isolation** via slots/reservations.
- Net effect: teams get **Dremel-class speed** without running infra, turning the research win into a **day-to-day advantage**.



Ask anything

+ Thinking ▾

0

ChatGPT can make mistakes. Check important info.