Questions d'entretien Data Architecture — Réponses et trade-offs

1. Fondamentaux et comparaisons

Data Warehouse vs Data Lake vs Lakehouse : quand choisir quoi ?

Réponse. Le warehouse stocke des données structurées, modélisées, optimisées pour le SQL analytique (Snowflake, BigQuery, Redshift). Le lake stocke tout en brut, à bas coût, sans schéma imposé à l'écriture (S3 + Parquet). Le lakehouse ajoute au lake des tables transactionnelles ACID (Delta, Iceberg) pour obtenir les garanties du warehouse sur le stockage du lake.

Trade-offs.

- Warehouse : performance et simplicité SQL, mais coût de stockage élevé et rigidité — il faut modéliser avant de charger, et les données non structurées (images, logs bruts) n'y ont pas leur place.
- Lake : coût minimal et flexibilité totale, mais risque de "data swamp" (données ingérées sans gouvernance, inutilisables), pas de transactions, performances médiocres sans optimisation.
- Lakehouse : un seul stockage pour BI et ML, mais écosystème plus jeune, complexité opérationnelle (compaction, vacuum, gestion des métadonnées) que le warehouse managé cache.

Règle de décision. BI classique avec équipe SQL → warehouse. ML + gros volumes hétérogènes + budget serré → lakehouse. Le lake pur seul est rarement une cible finale aujourd'hui, c'est une couche d'un lakehouse.

Analogie. Warehouse = bibliothèque avec livres classés par cote. Lake = garde-meuble où on entasse des cartons. Lakehouse = garde-meuble avec un inventaire tenu à jour et des étagères là

où c'est utile.

ETL vs ELT : pourquoi l'ELT s'est imposé ?

Réponse. En ETL, la transformation se fait sur un serveur intermédiaire avant chargement (Informatica, Talend). En ELT, on charge le brut dans le warehouse et on transforme en SQL dedans (dbt).

Pourquoi le basculement. Le compute cloud (MPP) est devenu élastique et bon marché : il est plus simple de transformer là où sont les données que de maintenir un moteur de transformation séparé. En plus, garder le brut permet de rejouer les transformations quand la logique change.

Trade-offs de l'ELT.

- Pour : brut conservé (auditabilité, rejeu), transformations en SQL versionnable (dbt), pas d'infra intermédiaire.
- Contre : coût compute du warehouse peut exploser si les transformations sont mal écrites ; données sensibles chargées en brut → il faut gérer PII/RGPD *dans* le warehouse (masking, colonnes chiffrées) au lieu de filtrer avant ; certaines transformations lourdes (parsing complexe, ML) restent mieux en Spark/Python.

OLTP vs OLAP

Réponse. OLTP : beaucoup de petites transactions (lecture/écriture de lignes individuelles), modèle normalisé, stockage orienté lignes (PostgreSQL, MySQL). OLAP : peu de requêtes mais lourdes (agrégations sur des millions de lignes, peu de colonnes), modèle dénormalisé, stockage colonnaire.

Implication design. Ne jamais faire tourner l'analytique sur la base OLTP de production : les scans analytiques verrouillent/ralentissent les transactions. D'où la réplication vers un système OLAP (CDC → warehouse).

Trade-off à mentionner. Certains systèmes promettent l'HTAP (hybride, ex : SingleStore, AlloyDB) — pratique pour éviter un pipeline, mais on paie en coût et on perd la spécialisation. Pour la plupart des cas, séparer reste plus simple et moins cher.

Star Schema vs Snowflake Schema

Réponse. Étoile : table de faits + dimensions dénormalisées (une table par dimension). Flocon : dimensions normalisées en sous-tables (dim_produit → dim_categorie → dim_famille).

Trade-offs.

- Étoile : requêtes simples (peu de jointures), performances meilleures, compréhensible par le métier. Coût : redondance dans les dimensions (négligeable, les dimensions sont petites).
- Flocon : moins de redondance, mises à jour de référentiels plus propres. Coût : jointures en cascade, requêtes plus complexes, gains de stockage insignifiants à l'ère du colonnaire compressé.

Position à défendre. Aujourd'hui, l'étoile gagne presque toujours : le stockage est bon marché, le temps humain non. Le flocon se justifie surtout si un référentiel est partagé et mis à jour fréquemment.

Data Mesh : quand décentraliser ?

Réponse. Le mesh transfère la propriété des données aux domaines métier, qui exposent des "data products" avec des contrats, sur une plateforme self-service, sous gouvernance fédérée.

Trade-offs.

- Pour : supprime le goulot d'étranglement de l'équipe data centrale, les domaines connaissent leurs données mieux que quiconque.
- Contre : exige une maturité forte partout (chaque domaine doit avoir des compétences data), duplication d'efforts, risque d'incohérence des définitions métier entre domaines, coût de la plateforme self-service.

Règle de décision. Le mesh est une réponse à un problème d'*échelle organisationnelle*, pas technique. En dessous de ~50-100 personnes dans la data ou de plusieurs domaines réellement autonomes, une équipe centrale avec un bon warehouse est plus efficace. Répondre "data mesh" pour une PME est un signal d'alarme en entretien.

Batch vs Streaming, Lambda vs Kappa

Réponse. Batch : traitement par lots planifiés (simple, économe, latence en heures). Streaming : traitement au fil de l'eau (latence en secondes, complexité supérieure). Lambda : deux pipelines parallèles (batch pour l'exactitude, streaming pour la fraîcheur) fusionnés à la lecture. Kappa : un seul pipeline streaming, le batch devient un rejeu du log.

Trade-offs.

- Lambda : résultats exacts (le batch corrige le streaming), mais deux bases de code à maintenir pour la même logique — coût de maintenance et risque de divergence.

- Kappa : une seule logique, mais tout le monde n'a pas besoin de streaming, et le rejeu de gros historiques via Kafka est coûteux.

Question à retourner en entretien. "Quelle latence le métier exige-t-il réellement ?" 90 % des besoins déclarés "temps réel" sont satisfaits par du micro-batch de 5-15 min, dix fois moins cher à opérer. C'est LE trade-off à verbaliser : latence contre complexité opérationnelle et coût.

---

2. Modélisation

SCD Type 1, 2, 3 et implémentation du Type 2

Réponse.

- Type 1 : on écrase l'ancienne valeur. Pas d'historique.
- Type 2 : on crée une nouvelle ligne à chaque changement, avec `valid_from`, `valid_to` (ou `end_date = 9999-12-31`), un flag `is_current`, et une clé de substitution (surrogate key) par version.
- Type 3 : une colonne `previous_value` — historique limité à un pas.

Implémentation Type 2 (à décrire). Comparer le snapshot source à la dimension courante (hash des colonnes suivies), fermer les lignes modifiées (`valid_to = now`, `is_current = false`), insérer les nouvelles versions. En dbt : `snapshots`. La table de faits référence la surrogate key, donc chaque fait pointe vers la version de la dimension valide *au moment du fait*.

Trade-offs. Type 2 donne l'exactitude historique mais fait grossir la dimension et complexifie les jointures (toujours filtrer sur la période ou `is_current`). Type 1 est trivial mais réécrit l'histoire — un rapport de 2023 relancé aujourd'hui donnera des chiffres différents. Le choix dépend de la question métier : "quel était le segment du client au moment de l'achat ?" → Type 2 obligatoire.

Types de tables de faits

Réponse.

- Fait transactionnel : une ligne par événement (une vente). Grain le plus fin, table la plus volumineuse.
- Snapshot périodique : une ligne par entité par période (solde de compte chaque fin de journée). Volume prévisible, parfait pour les états.

- Snapshot cumulatif : une ligne par processus avec des jalons mis à jour (commande : date_commande, date_paiement, date_livraison). Idéal pour analyser des durées de cycle, mais nécessite des updates.

Trade-off du grain. Toujours modéliser au grain le plus fin possible : on peut agréger, jamais désagréger. Le coût est le volume — qu'on gère par partitionnement et tables d'agrégats en aval, pas en dégradant le grain source.

Kimball vs Inmon

Réponse. Inmon : construire d'abord un warehouse d'entreprise normalisé (3NF), puis en dériver des data marts. Kimball : construire directement des modèles dimensionnels par processus métier, reliés par des dimensions conformes (bus architecture).

Trade-offs.

- Inmon : cohérence d'entreprise forte, mais time-to-value long — le métier attend des mois avant le premier dashboard.
- Kimball : valeur livrée vite, itératif, mais risque de silos si les dimensions conformes ne sont pas disciplinées.

Position moderne. L'approche dominante est un hybride : couche brute (le lake joue le rôle du "warehouse d'entreprise" d'Inmon, sans la normalisation), puis modèles dimensionnels Kimball dans la couche Gold. Data Vault s'insère entre les deux quand l'auditabilité et l'intégration multi-sources à grande échelle priment (banque, assurance) — au prix d'une complexité de modélisation nettement supérieure et de requêtes illisibles sans couche de présentation.

---

3. Design de bout en bout (mise en situation)

"Conçois l'architecture data d'un e-commerce"

Méthode attendue — commencer par les questions, pas par les technos :

1. Quels consommateurs ? (BI finance, dashboards temps réel ops, ML de reco, export CRM)

2. Quels volumes et vélocité ? (commandes/jour, événements web/s)
3. Quelle latence par usage ? (finance : J+1 suffit ; stock : minutes)
4. Contraintes ? (équipe, budget, cloud imposé, RGPD)

Réponse type (à adapter) :

- **Sources** : base OLTP des commandes, événements web/app (clickstream), CRM, référentiels produits.
- **Ingestion** : CDC (Debezium/Fivetran) pour l'OLTP — évite les extractions full qui chargent la prod ; Kafka ou équivalent managé pour le clickstream ; connecteurs SaaS pour le CRM.
- **Stockage** : lakehouse (S3 + Delta/Iceberg) ou warehouse cloud selon la maturité — architecture médaillon Bronze/Silver/Gold.
- **Transformation** : dbt pour le SQL (Silver → Gold), Spark si parsing lourd ou ML features.
- **Orchestration** : Airflow/Dagster.
- **Exposition** : Gold → outil BI ; reverse ETL vers le CRM ; feature store si ML.
- **Transverse** : catalogue + lineage, tests de qualité (dbt tests), monitoring, RBAC.

Trade-offs à verbaliser spontanément : "Je mets le clickstream en streaming parce que le volume rend le batch quotidien fragile, mais les commandes en CDC micro-batch 15 min car personne n'a besoin de mieux — le vrai temps réel doublerait le coût opérationnel."

Migration on-premise → cloud : big bang vs incrémental ?

Réponse. Presque toujours incrémental : migrer par domaine ou par pipeline, faire tourner les deux systèmes en parallèle (dual-run) avec réconciliation des chiffres, basculer les consommateurs progressivement, puis décommissionner.

Trade-offs.

- Incrémental : risque maîtrisé, valeur livrée tôt, mais période de double coût (deux infras) et de double maintenance qui peut s'éterniser si personne ne pilote le décommissionnement.
- Big bang : pas de double coût prolongé, mais risque énorme — si les chiffres divergent le jour J, tout le reporting de l'entreprise est en cause.
- Lift-and-shift (copier tel quel) vs re-architecture : le lift-and-shift va vite mais transporte la dette technique et coûte cher en cloud (les patterns on-prem sont inadaptés au pricing cloud) ; la re-architecture est l'occasion de nettoyer mais double la durée. Compromis courant : lift-and-shift des données, ré-écriture des pipelines critiques.

Idempotence et reprise sur échec

Réponse. Un pipeline idempotent produit le même résultat qu'on l'exécute une ou dix fois. Techniques : écriture par remplacement de partition (`INSERT OVERWRITE` sur la partition du jour) plutôt qu'append ; MERGE sur clé métier ; découpage des jobs en étapes atomiques ; écriture transactionnelle (Delta/Iceberg) pour éviter les états à moitié écrits.

Pourquoi c'est crucial. Les jobs échouent et les orchestrateurs relancent. Sans idempotence, chaque relance duplique des lignes — et on le découvre des semaines plus tard dans les chiffres du CFO.

Trade-off. L'overwrite de partition est simple et robuste mais recalcule toute la partition (coût compute) ; le MERGE est chirurgical mais plus cher par ligne et exige une clé fiable. Pour des late arriving data qui touchent de vieilles partitions, le MERGE devient obligatoire.

Schema evolution sans casser les consommateurs

Réponse. Règles : les ajouts de colonnes sont sûrs, les suppressions et renommages cassent. Stratégies : data contracts avec le producteur (schéma validé en CI), schema registry sur Kafka avec règles de compatibilité (backward : les nouveaux consommateurs lisent l'ancien format), versionner les changements cassants (v1/v2 en parallèle avec période de dépréciation), formats de table qui supportent l'évolution (Iceberg gère le renommage via des IDs de colonnes, pas les noms).

Trade-off. Bloquer les changements cassants en CI protège l'aval mais ralentit les équipes produit qui ne veulent pas se coordonner avec la data. Le data contract est autant un outil organisationnel que technique : il déplace la négociation *avant* le déploiement au lieu de l'incident *après*.

---

4. Performance

Optimiser une requête lente sur gros volume

Checklist de réponse :

1. **Lire moins de données** : partition pruning (filtrer sur la colonne de partition), format colonnaire (ne lire que les colonnes nécessaires), predicate pushdown, statistiques/z-ordering pour le data skipping.

2. **Joindre moins cher** : broadcast join si une table est petite, pré-agréger avant de joindre, colocaliser (bucketing) les grosses jointures récurrentes.
3. **Matérialiser** : tables d'agrégats ou vues matérialisées pour les requêtes répétées.
4. **Dimensionner** : en dernier recours seulement — scaler du compute sur une requête mal écrite, c'est payer pour cacher le problème.

Trade-off central. Chaque optimisation de lecture se paie à l'écriture ou en stockage (agrégats à maintenir, compaction, tri). On optimise pour le pattern de lecture *dominant*, pas pour toutes les requêtes.

Partitionnement vs bucketing

Réponse. Partitionnement : découpage physique en répertoires par valeur (souvent la date) → élimine des fichiers entiers à la lecture. Bucketing : répartition par hash d'une clé en N buckets fixes → accélère jointures et agrégations sur cette clé en évitant le shuffle.

Trade-offs. Partitionner sur une colonne à trop forte cardinalité (user_id) crée des millions de petites partitions — catastrophe métadonnées. Règle : partitionner sur ce qu'on filtre (date), bucketer sur ce qu'on joint (id client). Le bucketing fige N à l'écriture : si le volume décuple, il faut réécrire.

Le problème des small files

Réponse. Des milliers de petits fichiers (issus de streaming ou de partitions trop fines) tuent les performances : chaque fichier a un coût fixe d'ouverture (métadonnées, listing S3), et les moteurs parallélisent par fichier — 10 000 fichiers de 10 KB sont bien plus lents à lire qu'un fichier de 100 MB.

Solutions. Compaction périodique (`OPTIMIZE` en Delta, rewrite en Iceberg), régler la taille cible (~128 MB-1 GB), réduire la fréquence des micro-batchs d'écriture.

Trade-off. La compaction consomme du compute et peut entrer en conflit avec les écritures concurrentes ; écrire des fichiers plus gros augmente la latence de disponibilité des données. Encore latence contre coût.

Spark : fonctionnement interne et data skew

Réponse courte attendue. Spark construit un DAG d'opérations, évalué paresseusement (lazy) : rien ne s'exécute avant une action. Le plan est découpé en stages aux frontières de shuffle (échange de données entre nœuds, déclenché par groupBy/join sur clé). Chaque stage s'exécute en tâches parallèles, une par partition. Si une partition ne tient pas en mémoire → spill sur disque → lenteur.

Data skew. Quand une clé concentre une part disproportionnée des données (le client "null", un produit ultra-populaire), la tâche qui la traite devient le goulot : 199 tâches finissent en 1 min, la 200e tourne 2 h.

Corrections. Salting (suffixe aléatoire sur la clé chaude pour l'éclater, puis ré-agrégation), broadcast join pour éviter le shuffle, isoler/filtrer les clés dégénérées (nulls), AQE (Adaptive Query Execution) qui splitte automatiquement les partitions obèses.

Trade-off du salting. Il éclate la clé chaude mais ajoute une étape d'agrégation et complexifie le code — à réserver aux skews avérés, pas en préventif.

---

5. Gouvernance, qualité, RGPD

Garantir la qualité des données

Réponse. Tests à chaque couche : contraintes à l'ingestion (schéma, nullabilité), tests dbt (unicité, not null, intégrité référentielle, valeurs acceptées), tests métier (le CA quotidien ne varie pas de ±50 % sans raison), observabilité en continu (Monte Carlo, Elementary : fraîcheur, volume, dérive de schéma).

Trade-off clé. Que faire quand un test échoue ? Bloquer le pipeline (les consommateurs préfèrent des données en retard que fausses — vrai pour la finance) ou alerter sans bloquer (vrai pour un dashboard produit où la fraîcheur prime) ? La réponse dépend du consommateur, et c'est exactement ce qu'un data contract doit spécifier. Dire "on bloque tout" ou "on alerte seulement" sans distinguer est une réponse faible.

RGPD dans un data lake (question piège)

Le piège. Les fichiers Parquet sont immuables et le droit à l'oubli exige la suppression. Supprimer une ligne = réécrire le fichier — inefficace à l'échelle, et impossible à tracer dans les archives.

Réponses solides :

1. **Formats transactionnels** : `DELETE` en Delta/Iceberg réécrit les fichiers concernés proprement — mais attention au time travel/vacuum : les anciennes versions doivent aussi être purgées (régler la rétention).
2. **Crypto-shredding** : chiffrer les PII avec une clé par personne ; supprimer la clé rend les données illisibles partout, y compris les backups. Élégant, mais gestion de clés à l'échelle non triviale.
3. **Pseudonymisation en amont** : séparer les PII dans un store dédié dès l'ingestion, ne faire circuler que des identifiants — le lake ne contient alors rien à supprimer.

Trade-off. La pseudonymisation en amont est la plus propre mais doit être conçue dès le départ ; le crypto-shredding se rattrape après coup mais ajoute une dépendance critique au key management.

---

6. Distribué et théorie

Théorème CAP

Réponse. En cas de partition réseau (P, inévitable en distribué), un système doit choisir entre cohérence (C : tous les nœuds voient la même donnée) et disponibilité (A : toute requête reçoit une réponse). Ce n'est pas un choix permanent mais un comportement *pendant la partition*.

Implication pratique. Un système bancaire choisit C (mieux vaut refuser une transaction que créer une incohérence de solde) ; un panier e-commerce choisit A (mieux vaut accepter la commande et réconcilier après). ACID vs BASE en découle : garanties transactionnelles strictes vs cohérence à terme.

Nuance qui distingue un senior. CAP est souvent surinvoqué : la plupart des trade-offs réels du quotidien sont latence vs cohérence en fonctionnement *normal* (PACELC), pas pendant les partitions.

Exactly-once dans Kafka

Réponse. At-least-once : le message est garanti livré mais peut l'être en double (le consommateur commit son offset après traitement ; s'il crashe entre les deux, il retraite). Exactly-once nécessite : producteur idempotent (déduplication par ID de séquence côté broker), transactions Kafka (écrire le résultat et commiter l'offset atomiquement), ou — le plus robuste — un consommateur idempotent en aval (MERGE sur clé, écriture par partition).

Trade-off. L'exactly-once transactionnel coûte en latence et en complexité, et il ne couvre que le périmètre Kafka : dès qu'on écrit vers un système externe, c'est l'idempotence de l'écriture finale qui compte. Réponse forte en entretien : "je préfère at-least-once + sinks idempotents que de dépendre des transactions de bout en bout."

---

7. Questions d'expérience — comment les structurer

"Décris une architecture que tu as conçue et ses trade-offs"

Structure gagnante (STAR orienté trade-offs) :

1. Contexte et contrainte dominante (volume ? budget ? équipe de 2 ? deadline ?)
2. Deux options réellement considérées.
3. Le choix et *ce qu'on a sacrifié en connaissance de cause*.
4. Ce que ça a donné, et ce que tu referais différemment.

Le point 3 est ce que l'intervieweur écoute : un candidat qui ne peut pas nommer ce que son architecture sacrifie n'a pas fait d'architecture, il a suivi un tutoriel.

"Build vs buy ?"

Grille de réponse. Buy (managé/SaaS) par défaut pour tout ce qui n'est pas différenciant (ingestion, orchestration, catalogue) : le coût caché du build est la maintenance perpétuelle, pas le développement initial. Build quand : besoin réellement spécifique, coût SaaS prohibitif à l'échelle, ou dépendance stratégique inacceptable (lock-in). Toujours chiffrer le coût humain du run, pas seulement la licence.

Les 3 réflexes transverses à montrer

1. **Partir du besoin, pas de l'outil.** Toute réponse de design commence par des questions sur consommateurs, volume, latence, budget.
2. **Nommer le sacrifice.** Chaque choix a un coût ; le verbaliser spontanément ("je gagne X, je paie Y") est le marqueur de séniorité.
3. **Le trio latence / coût / complexité.** La quasi-totalité des trade-offs en data architecture se ramène à ces trois axes. Quand tu hésites sur une réponse, raisonne sur ce triangle à voix haute.

Revision #2
Created 20 July 2026 16:28:19 by EMB
Updated 20 July 2026 16:28:41 by EMB