L'ENTREPRISE AGENTIQUE

VOLUME IV

Apache Iceberg

Le Lakehouse Moderne

André-Guy Bruneau

2026

Table des Matières

Chapitre IV.1 — Le Monde du Lakehouse Apache Iceberg 21
Introduction : L'Émergence d'un Nouveau Paradigme 21
De l'Entrepôt au Lac : Une Évolution Nécessaire 21
Le Contexte Historique 21
Les Limites de l'Entrepôt Traditionnel 22
L'Avènement et les Promesses du Lac de Données 22
Apache Hive : Une Première Tentative de Structure 23
La Genèse d'Apache Iceberg 23
Netflix et la Naissance d'un Standard 23
Le Concept de Format de Table Ouvert 24
Positionnement dans l'Écosystème 24
Le Paradigme du Data Lakehouse 26
Définition et Principes Fondateurs 26
Les Cinq Couches de l'Architecture Lakehouse 26
Avantages Stratégiques pour l'Entreprise 27
Apache Iceberg : Vue d'Ensemble Technique 28
Architecture à Trois Niveaux 28
Le Rôle Central des Métadonnées 28
Transactions ACID et Isolation 29
Évolution de Schéma 30
Partitionnement Masqué 30
Stratégies d'Écriture : Copy-on-Write et Merge-on-Read 31
Écosystème et Adoption Industrielle 32
Support Multi-Moteur 32
Convergence de l'Industrie 32
Adoption par les Géants Technologiques 33
Tendances d'Adoption par Secteur 33
Perspectives Canadiennes et Contexte Réglementaire 34
Exigences Réglementaires Distinctives 34
Souveraineté des Données 35
Écosystème de Partenaires Canadiens 36
Défis et Considérations 36
Complexité Opérationnelle 36
Courbe d'Apprentissage 37
Cas Où Iceberg N'est Pas Optimal 37
La Voie Vers l'Adoption 38
Conclusion : Préparer l'Avenir des Données d'Entreprise 38
L'Impératif de la Modernisation 39
Le Rôle du Lakehouse dans l'Entreprise Agentique 39
Perspectives Technologiques 2026-2030 39
Recommandations pour les Décideurs 40
Résumé 40
Évolution Architecturale 40
Origines et Genèse d'Iceberg 40
Architecture Technique 41

Écosystème et Adoption 41
Contexte Canadien 41
Considérations Pratiques 42
Chapitre IV.2 — Anatomie Technique d'Apache Iceberg 43
Introduction 43
L'Architecture en Trois Couches 44
Vue d'Ensemble de l'Architecture 44
Flux de Lecture d'une Table 44
La Couche de Catalogue 44
Rôle et Responsabilités 45
Types de Catalogues 45
La Spécification REST Catalog 45
La Couche de Métadonnées 46
Le Fichier metadata.json 46
Les Snapshots 46
Les Listes de Manifestes 47
Les Fichiers Manifestes 48
La Couche de Données 48
Formats de Fichiers Supportés 49
Configuration des Formats de Fichiers 49
Immutabilité des Fichiers 49
Organisation Physique 50
Tri des Données 50
Gestion des Schémas 50
Le Système d'Identifiants de Colonnes 50
Opérations d'Évolution de Schéma 51
Valeurs par Défaut 51
Promotions de Type Sûres 51
Schémas Imbriqués 52
Gestion du Partitionnement 52
Partitionnement Masqué 52
Transformations de Partition 53
Évolution du Partitionnement 53
Bonnes Pratiques de Partitionnement 54
Opérations au Niveau des Lignes 54
Copy-on-Write (COW) 55
Merge-on-Read (MOR) 55
Vecteurs de Suppression (Version 3) 56
Configuration des Stratégies 56
Statistiques et Optimisation des Requêtes 56
Statistiques de Manifeste 56
Filtres de Bloom 57
Fichiers Puffin et Statistiques Avancées 57
Élagage Multi-Niveau 58
Transactions et Concurrence 58
Isolation des Lectures 58
Concurrence Optimiste 58
Types de Conflits 59
Niveau d'Isolation Sérialisable 59

Les Tables de Métadonnées 59
Tables de Métadonnées Principales 60
Utilisation pour le Diagnostic 60
Le Processus de Commit en Détail 61
Anatomie d'un Commit 61
Retry et Backoff Exponentiel 62
Gestion des Fichiers Orphelins 62
Branches et Tags 62
Branches 62
Tags 63
Rétention des Branches et Tags 63
Intégration avec les Moteurs de Requête 63
Apache Spark 64
Trino 64
Apache Flink 64
Dremio 64
Considérations Avancées de Performance 65
Prolifération de Petits Fichiers 65
Accumulation de Snapshots 65
Fichiers de Suppression Non Compactés 66
Statistiques Obsolètes 66
Taille du Fichier de Métadonnées 67
Monitoring de la Santé des Tables 67
Résumé 67
Architecture en Trois Couches 68
Hiérarchie de Métadonnées 68
Système d'Identifiants de Colonnes 68
Partitionnement Masqué et Évolution 68
Stratégies de Modification au Niveau Lignes 69
Statistiques Multi-Niveaux 69
Concurrence Optimiste et Transactions 69
Branches et Tags 69
Implications Pratiques 69
Perspectives Techniques 70
Chapitre IV.3 — Mise en Pratique avec Apache Iceberg 71
IV.3.1 Configuration d'un environnement Apache Iceberg 71
Prérequis techniques 72
Architecture de l'environnement de développement 72
Fichier Docker Compose 73
Initialisation de l'environnement 74
Configuration du stockage MinIO 75
Validation de la connectivité 75
Résolution des problèmes courants 76
Comprendre l'architecture des fichiers Iceberg sur MinIO 77
IV.3.2 Création de tables Iceberg dans Spark 77
Accès à l'environnement Spark 78
Création de l'espace de noms et de la table 78
Insertion de données 80
Validation et exploration des données 80

Exploration des snapshots et du voyage dans le temps 81
Démonstration de l'évolution de schéma 82
Création d'une table de transactions 83
Opérations avancées : MERGE INTO 85
Exploration des fichiers de métadonnées 85
Utilisation des branches Nessie (optionnel) 86
IV.3.3 Lecture des tables Iceberg dans Dremio 87
Configuration initiale de Dremio 87
Connexion au catalogue Nessie 88
Exploration des données via l'interface SQL 88
Fonctionnalités Iceberg dans Dremio 89
Création de vues virtuelles 89
Configuration des réflexions de données 90
Connexion à des sources externes 91
Requêtes fédérées et virtualisation 91
Gestion des droits d'accès 92
Optimisation des requêtes avec EXPLAIN 92
Profils de requête et métriques 93
IV.3.4 Création d'un tableau de bord BI 93
Option 1 : Apache Superset 93
Option 2 : Connexion Power BI 96
Option 3 : Connexion Tableau 97
Considérations pour la production 98
Bonnes pratiques pour les tableaux de bord Lakehouse 99
IV.3.5 Résumé 101
Compétences acquises 101
Architecture mise en œuvre 101
Points clés à retenir 102
Commandes et syntaxes essentielles 103
Prochaines étapes recommandées 104
Comparaison des approches d'interrogation 104
Transition vers la Partie 2 104
Nettoyage de l'environnement de développement 105
Ressources complémentaires 105
Chapitre IV.4 — Préparer Votre Passage à Apache Iceberg 107
IV.4.1 Réalisation de l'audit de votre plateforme 107
Objectifs de l'audit 107
Les six dimensions de l'audit 108
Méthodes de collecte d'information 109
Livrables de l'audit 110
Facteurs de succès de l'audit 111
IV.4.2 L'audit de la Banque Hamerliwa en action 112
Profil de la Banque Hamerliwa 112
Contexte de l'audit 112
Constats de l'audit — Dimension Architecture 113
Constats de l'audit — Dimension Données 113
Constats de l'audit — Dimension Flux 114
Constats de l'audit — Dimension Consommation 114
Constats de l'audit — Dimension Gouvernance 115

Constats de l'audit — Dimension Coûts 115
Synthèse de l'audit Hamerliwa 116
IV.4.3 De l'audit aux exigences 116
Structure des exigences 117
Exigences fonctionnelles pour Hamerliwa 117
Exigences non fonctionnelles pour Hamerliwa 118
Exigences de migration pour Hamerliwa 119
Exigences de gouvernance pour Hamerliwa 120
Matrice de traçabilité 120
IV.4.4 Plan architectural et présentation itinérante 121
Vision architecturale de haut niveau 121
Diagramme d'architecture cible 122
Feuille de route de migration 123
La présentation itinérante 123
Gestion des objections 125
Stratégie de gestion du changement 126
Documentation et gouvernance du plan 127
IV.4.5 Résumé 127
Concepts clés 127
Livrables de la phase de préparation 128
Facteurs de succès 128
Prochaines étapes 128
Commandes et outils essentiels 129
Transition vers le chapitre suivant 130
Chapitre IV.5 - Sélection de la Couche de Stockage 131
Introduction 131
Fondamentaux du Stockage Objet pour Apache Iceberg 131
Anatomie du Stockage Objet 131
Exigences Spécifiques d'Iceberg 132
Séparation Calcul-Stockage : Implications Architecturales 132
Panorama des Solutions de Stockage 133
Stockage Infonuagique Public 133
Stockage sur Site et Hybride 135
Solutions Hybrides et Multi-nuage 136
Critères de Sélection 136
Performance et Latence 136
Coûts et Modèle Économique 137
Résidence et Souveraineté des Données 138
Intégration avec l'Écosystème Existant 139
Configuration Optimale par Plateforme 140
Amazon S3 : Configuration de Référence 140
Azure ADLS Gen2 : Configuration Optimale 142
MinIO : Déploiement de Production 143
Stratégies de Migration 145
Migration depuis HDFS 145
Migration depuis un Data Warehouse 146
Migration Multi-nuage 146
Haute Disponibilité et Reprise après Sinistre 147
Architectures de Résilience 147

Stratégies de Basculement 148
Sauvegarde et Restauration Iceberg 148
Optimisation des Performances de Stockage 149
Dimensionnement des Fichiers 149
Stratégies de Compaction 150
Mise en Cache et Accélération 150
Études de Cas Canadiennes 151
Secteur Financier : Coopérative de Services Financiers 151
Secteur Énergie : Société d'État Hydroélectrique 152
Secteur Santé : Réseau de Recherche Clinique 152
Matrice de Décision 153
Tableau Comparatif des Solutions 153
Arbre de Décision 153
Recommandations par Profil 153
Conclusion 154
Résumé 155
Chapitre IV.6 - Architecture de la Couche d'Ingestion 156
Introduction 156
Patterns Fondamentaux d'Ingestion 156
Taxonomie des Modes d'Ingestion 156
Évolution vers le Streaming Lakehouse 157
Sémantiques de Livraison 157
Architecture de Référence 158
Vue d'Ensemble 158
Composantes Clés 159
Dimensionnement Initial 159
Ingestion Batch avec Apache Spark 160
Configuration Spark-Iceberg 160
Patterns d'Écriture Batch 160
Optimisation des Extractions JDBC 161
Ingestion Streaming avec Apache Kafka 163
Architecture Kafka-Iceberg 163
Configuration Kafka Connect Iceberg Sink 163
Apache Flink pour le Streaming Temps Réel 164
Spark Structured Streaming 165
Change Data Capture (CDC) 166
Fondamentaux du CDC 166
Debezium : La Référence Open Source 167
Traitement des Événements CDC dans Iceberg 168
Gestion des Suppressions 169
Ingestion de Fichiers 170
Patterns d'Ingestion de Fichiers 170
Détection et Déclenchement 171
Traitement de Formats Variés 172
Qualité des Données à l'Ingestion 173
Validation en Ligne 173
Déduplication 175
Gestion des Erreurs et Quarantaine 175
Orchestration des Pipelines 177

Apache Airflow pour l'Orchestration 177
Patterns d'Orchestration Avancés 179
Performance et Optimisation 181
Dimensionnement des Pipelines 181
Optimisation des Écritures Iceberg 181
Monitoring et Observabilité 182
Études de Cas Canadiennes 183
Secteur Télécommunications : Opérateur National 183
Secteur Commerce de Détail : Chaîne Nationale 183
Secteur Public : Agence Gouvernementale 184
Matrice de Décision Technologique 185
Choix du Moteur d'Ingestion 185
Arbre de Décision 185
Conclusion 185
Résumé 186
Chapitre IV.7 - Implémentation de la Couche de Catalogue 188
Introduction 188
Fondamentaux du Catalogue Iceberg 188
Rôle et Responsabilités 188
Architecture Découplée 189
Interface Catalog d'Iceberg 190
Panorama des Implémentations de Catalogue 190
Hive Metastore 190
REST Catalog 191
AWS Glue Data Catalog 192
Azure Purview et Unity Catalog 193
Catalogues Commerciaux 193
Nessie : Le Catalogue Git-like 194
Implémentation du REST Catalog 194
Architecture de Déploiement 194
Déploiement avec Polaris (Incubateur Apache) 195
Configuration Client 198
Implémentation AWS Glue Catalog 199
Configuration et Déploiement 199
Intégration avec Lake Formation 202
Implémentation Nessie 203
Concepts et Architecture 203
Déploiement Nessie 204
Flux de Travail Git-like 205
Fédération Multi-Catalogue 207
Architecture de Fédération 207
Configuration Trino Multi-Catalogue 208
Dremio comme Couche de Fédération 208
Sécurité et Gouvernance du Catalogue 209
Modèles d'Authentification 209
Contrôle d'Accès Granulaire 210
Audit et Traçabilité 211
Migration entre Catalogues 212
Stratégies de Migration 212

Migration Hive Metastore vers REST Catalog 212
Migration Glue vers REST Catalog 215
Études de Cas Canadiennes 216
Secteur Assurance : Assureur National 216
Secteur Minier : Société d'Exploration 216
Secteur Public : Ministère Provincial 217
Bonnes Pratiques et Recommandations 218
Convention de Nommage 218
Haute Disponibilité 218
Surveillance et Alertes 219
Matrice de Décision 220
Conclusion 221
Résumé 221
Chapitre IV.8 - Conception de la Couche de Fédération 223
Introduction 223
Fondamentaux de la Fédération de Données 223
Définition et Objectifs 223
Architecture de Fédération pour Iceberg 224
Patterns de Requêtes Fédérées 225
Moteurs de Requête Fédérée 226
Trino : Le Standard Open Source 226
Dremio : La Plateforme Lakehouse Unifiée 229
Apache Spark SQL 231
Starburst Galaxy et Enterprise 232
Comparaison des Moteurs 233
Configuration Avancée des Connecteurs 233
Connecteur Iceberg Optimisé 233
Connecteurs Sources Externes 234
Configuration Multi-Catalogue 235
Optimisation des Performances 235
Predicate Pushdown et Projection Pushdown 235
Stratégies de Jointure 236
Gestion du Cache 237
Dimensionnement et Scaling 238
Couche Sémantique et Virtualisation 239
Concepts de la Couche Sémantique 239
Implémentation avec Dremio 240
Implémentation avec DBT et Trino 241
Sécurité et Gouvernance 244
Contrôle d'Accès Unifié 244
Masquage et Filtrage des Données 245
Audit et Traçabilité 246
Intégration avec les Outils BI 248
Power BI et Microsoft Fabric 248
Tableau 249
Superset et Metabase 249
Études de Cas Canadiennes 250
Secteur Bancaire : Institution Financière Majeure 250
Secteur Commerce : Détaillant Omnicanal 251

Secteur Santé : Réseau Hospitalier Provincial 251
Secteur Énergie : Producteur d'Électricité 252
Bonnes Pratiques et Recommandations 252
Architecture de Référence 252
Checklist de Déploiement 253
Matrice de Décision 254
Conclusion 254
Résumé 255
Chapitre IV.9 - Comprendre la Couche de Consommation 257
Introduction 257
Taxonomie des Consommateurs de Données 257
Profils d'Utilisateurs 257
Patterns d'Accès Caractéristiques 258
Exigences par Profil 259
Consommation pour l'Intelligence d'Affaires 260
Intégration Power BI 260
Intégration Tableau 261
Self-Service Analytics 262
Consommation pour la Science des Données 264
Accès Notebook et DataFrame 264
Feature Stores et Iceberg 265
MLOps et Intégration Continue 267
Consommation Applicative 269
APIs de Données 269
Accès Temps Réel et Faible Latence 272
Microservices et Architecture Événementielle 275
Patterns d'Accès Avancés Iceberg 276
Time Travel pour Analyse Historique 276
Branches pour Environnements Isolés 278
Lectures Incrémentielles 279
Gouvernance de la Consommation 281
Quotas et Limites de Ressources 281
Contrôle d'Accès Granulaire 282
Monitoring et Observabilité 283
Études de Cas Canadiennes 284
Secteur Manufacturier : Équipementier Automobile 284
Secteur Assurance : Mutuelle Pancanadienne 285
Secteur Public : Agence Provinciale de Transport 286
Secteur Détail : Épicier Québécois 287
Bonnes Pratiques et Recommandations 288
Architecture de Référence par Profil 288
Checklist de Mise en Production 288
Matrice de Décision Outils 289
Conclusion 289
Résumé 290
Chapitre IV.10 - Maintenir un Lakehouse Iceberg en Production 292
Introduction 292
Anatomie de la Maintenance Iceberg 292
Comprendre l'Accumulation des Fichiers 292

Cycle de Vie des Objets Iceberg 293
Opérations de Maintenance Essentielles 294
Compaction des Fichiers de Données 295
Principes et Stratégies 295
Procédure de Compaction avec Spark 296
Compaction avec Tri (Sort Compaction) 297
Stratégies de Compaction par Type de Table 298
Gestion des Snapshots 299
Expiration des Snapshots 299
Nettoyage des Fichiers Orphelins 300
Réécriture des Fichiers de Manifeste 302
Monitoring et Observabilité 303
Métriques Essentielles 303
Dashboard Grafana pour Lakehouse 304
Collecte Automatisée des Métriques 307
Automatisation des Opérations 308
Orchestration avec Apache Airflow 308
Maintenance Événementielle 312
Gestion des Incidents 313
Procédures de Diagnostic 313
Étape 2: Vérifier les statistiques de table 313
Étape 3: Analyser le plan de requête 314
Actions correctives: 314
Récupération après Incident 315
Sauvegarde et Récupération 317
Stratégies de Sauvegarde 317
Plan de Reprise après Sinistre 319
Optimisation Continue 321
Analyse des Patterns de Requêtes 321
Ajustement Automatique des Paramètres 322
Études de Cas Canadiennes 323
Secteur Financier : Banque d'Investissement 323
Secteur Télécommunications : Opérateur Mobile 324
Secteur Commerce : Détaillant Alimentaire 325
Secteur Public : Agence Statistique 325
Bonnes Pratiques et Recommandations 326
Checklist de Maintenance 326
Matrice de Décision 327
Anti-Patterns à Éviter 327
Conclusion 328
Résumé 328
Chapitre IV.11 - OPÉRATIONNALISER APACHE ICEBERG 330
Introduction 330
IV.11.1 Orchestration du Lakehouse 331
Le rôle de l'orchestration dans un lakehouse moderne 331
Panorama des orchestrateurs pour les lakehouses 331
Comparaison des orchestrateurs pour les lakehouses 332
Critères de sélection d'un orchestrateur 333
Modèles d'orchestration pour Apache Iceberg 334

Orchestrer les opérations de maintenance Iceberg 335
Infrastructure as Code pour l'orchestration 337
Gestion des environnements multiples 338
Bonnes pratiques d'orchestration 338
IV.11.2 Audit du Lakehouse 339
Exigences d'audit dans un contexte réglementaire 339
Architecture d'audit pour un lakehouse Iceberg 340
Intégration de la qualité des données dans l'audit 341
Architecture de collecte des journaux d'audit 343
Exploiter les tables de métadonnées pour l'audit 344
Observabilité et alertes 344
Intégration avec les plateformes d'observabilité 347
Conformité et rapports réglementaires 349
IV.11.3 Récupération Après Sinistre 350
Définir les objectifs de récupération 350
Capacités natives d'Iceberg pour la récupération 350
Scénarios de récupération et solutions 351
Stratégie de sauvegarde pour un lakehouse Iceberg 352
Configuration de la réplication S3 Cross-Region 353
Création d'un inventaire DR complet 354
Automatisation des tests de récupération 356
Plan de récupération après sinistre 360
Escalade 361
Considérations de coûts pour la DR 364
IV.11.4 Résumé 365
Orchestration du lakehouse 365
Audit du lakehouse 365
Récupération après sinistre 366
Principes directeurs 366
Points clés à retenir 366
Références 367
Chapitre IV.12 - L'ÉVOLUTION VERS LE STREAMING LAKEHOUSE 368
Introduction 368
IV.12.1 De l'Architecture Lambda au Streaming Lakehouse 368
Les Origines : Le Défi du Big Data 368
L'Architecture Lambda : Une Réconciliation Imparfaite 369
Les Limites Fondamentales de Lambda 370
L'Architecture Kappa : La Simplification par le Flux 370
L'Émergence du Data Lakehouse 371
Le Streaming Lakehouse : La Convergence 372
Comparaison des Architectures 373
Le Pattern « Shift Left » 373
Couches Bronze, Silver, Gold dans le Streaming Lakehouse 375
Gestion du Change Data Capture (CDC) 376
Architecture Delta et Streamhouse 378
IV.12.2 Le Rôle de Confluent et Kafka 378
Apache Kafka comme Système Nerveux Central 378
L'Écosystème Confluent 379
Tableflow : La Passerelle vers le Lakehouse 379

Patterns d'Intégration Kafka-Iceberg 380
Le Flink Dynamic Iceberg Sink 382
Intégration avec le Schema Registry 383
Gestion de l'Évolution de Schéma 384
Intégration avec les Catalogues Externes 385
Traitement Pré-Lakehouse avec Flink 386
Garanties de Livraison et Exactly-Once 388
Considérations de Coûts 389
Considérations de Performance 390
Observabilité du Pipeline Streaming-Lakehouse 390
L'Approche Lakehouse-Native avec Ursa 392
Considérations pour le Contexte Canadien 392
Architecture de Référence pour le Canada 396
Feuille de Route pour la Migration 397
Résolution des Problèmes Courants 399
Gestion des Données Tardives 401
IV.12.3 Résumé 402
L'Évolution Architecturale 402
Le Rôle Central de Kafka et Confluent 403
Patterns d'Implémentation 403
Considérations Opérationnelles 403
Recommandations Stratégiques 403
Références 404
Chapitre IV.13 - SÉCURITÉ, GOUVERNANCE ET CONFORMITÉ DU LAKEHOUSE 405
Introduction 405
IV.13.1 Architecture de Sécurité Multicouche 405
Le Défi de la Sécurité Distribuée 405
Sécurité du Stockage Objet 406
Le Chiffrement Natif Iceberg 409
Sécurité au Niveau du Catalogue 410
Project Nessie : Git pour les Données 413
Gestion des Clés et Rotation 415
IV.13.2 Contrôles d'Accès Granulaires 417
Les Niveaux de Contrôle d'Accès 417
Row-Level Security (RLS) 418
Column-Level Security et Masquage Dynamique 419
Vues Dynamiques Sécurisées 420
Patterns de Masquage Avancés 421
Tableaux de Bord de Gouvernance 424
Qualité des Données 428
Catalogage et Découverte des Données 430
IV.13.4 Conformité Réglementaire au Canada 431
La Loi 25 du Québec 431
La LPRPDE Fédérale 435
Conformité Sectorielle (BSIF, SOX) 435
IV.13.5 Audit et Surveillance 437
Journalisation des Accès 437
Alertes de Sécurité 439
Réponse aux Incidents de Sécurité 440

IV.13.6 Études de Cas Canadiennes 446
IV.13.7 Résumé 447
Architecture de Sécurité Multicouche 447
Contrôles d'Accès Granulaires 448
Gouvernance des Données 448
Conformité Canadienne 448
Audit et Réponse aux Incidents 448
Recommandations Stratégiques 449
Références 449
Chapitre IV.14 - L'Intégration avec Microsoft Fabric et Power BI 451
Introduction 451
IV.14.1 OneLake Shortcuts et Virtualisation 451
L'Architecture OneLake : Le Lac de Données Unifié 451
Les Raccourcis OneLake : Virtualisation sans Duplication 452
La Virtualisation des Métadonnées : Le Pont Iceberg-Delta 453
Configuration de l'Intégration Iceberg vers Fabric 454
La Conversion Delta vers Iceberg : Bidirectionnalité Complète 455
Types de Raccourcis et Configurations Avancées 456
Configuration de l'Authentification pour les Raccourcis Externes 457
Considérations de Performance pour les Raccourcis 457
Sécurité OneLake : Gouvernance Unifiée 458
IV.14.2 Power BI Direct Lake : Latence et Performance 459
Comprendre les Modes de Stockage Power BI 459
L'Architecture Direct Lake 460
Le Mécanisme de Chargement à la Demande 460
V-Order : L'Optimisation Propriétaire pour Direct Lake 461
Direct Lake pour les Tables Iceberg Virtualisées 461
Configuration d'un Modèle Sémantique Direct Lake 462
Gestion du Repli DirectQuery 463
Power BI Embedded avec Direct Lake 463
Considérations de Coûts et de Capacité 464
Le Partenariat Microsoft-Snowflake : Interopérabilité Native 465
La Fonctionnalité de Miroir (Mirroring) Snowflake 466
Limitations et Contraintes Actuelles 467
Dépannage et Résolution des Problèmes Courants 468
IV.14.3 Patterns d'Architecture pour l'Intégration Fabric-Iceberg 469
Pattern 1 : Lakehouse Hybride Multi-Format 469
Pattern 2 : Fabric comme Couche de Consommation Unifiée 470
Pattern 3 : Migration Progressive vers Fabric Natif 471
Intégration avec le Streaming Lakehouse 472
Intégration avec les Agents de Données IA et Copilot 473
IV.14.4 Recommandations et Bonnes Pratiques 475
Choix de l'Architecture d'Intégration 475
Optimisation des Performances 475
Gouvernance et Sécurité 475
Surveillance et Opérations 476
Considérations de Continuité d'Activité 476
Évolutions Futures et Feuille de Route 477
Résumé 477

Chapitre IV.15 — Contexte Canadien et Études de Cas 480
Introduction 480
IV.15.1 Introduction au Contexte Canadien 480
Le Paysage Numérique Canadien 480
Cadre Réglementaire et Conformité 481
Infrastructure Infonuagique au Canada 483
IV.15.2 Étude de Cas : Banque Royale du Canada (RBC) 484
Contexte et Vision Stratégique 484
Architecture de Données Événementielle avec Confluent 484
Borealis AI et l'Infrastructure IA Privée 485
Aiden : La Plateforme de Trading Algorithmique 485
Initiative de Modernisation de la Fraude 486
Reconnaissance et Maturité IA 486
Leçons pour les Architectes Lakehouse 486
Architecture de Référence Inspirée de RBC 487
IV.15.3 Étude de Cas : Bell Canada 488
Contexte et Transformation Numérique 488
Partenariat Stratégique avec Google Cloud 488
Solution Network AI Ops 489
Transformation avec ServiceNow 489
Stratégie IA et Centres de Données 490
Bell Cyber et la Souveraineté des Données 490
Lakehouse pour les Télécommunications 490
Architecture de Données Télécommunications 491
IV.15.4 Souveraineté des Données et Infrastructure Régionale 492
Résidence versus Souveraineté 492
La CLOUD Act et ses Implications 493
Stratégies d'Atténuation des Risques 493
Patterns d'Architecture pour les Autres Secteurs Canadiens 494
Infrastructure Régionale Disponible 495
Considérations pour le Data Lakehouse 496
Le Cas des Environnements Multi-Cloud et Hybrides 497
IV.15.5 Recommandations pour les Architectes Canadiens 497
Considérations Économiques 497
Principes Directeurs 498
Patterns d'Architecture Recommandés 499
Checklist de Conformité Lakehouse 501
Défis de Mise en Œuvre et Solutions 501
Évolution du Paysage Canadien 502
Feuille de Route pour l'Adoption 503
Résumé 504
Chapitre IV.16 - Conclusion Finale et Perspectives 2026-2030 507
Introduction 507
Synthèse des Fondamentaux Iceberg 507
Les Piliers Architecturaux 507
Leçons Opérationnelles 508
Architecture de Référence Consolidée 509
État de l'Écosystème 2025-2026 509
Consolidation du Marché 509

Maturité Technologique 510
Défis Persistants 510
Tendances Technologiques 2026-2030 511
Intelligence Artificielle et Lakehouse 511
Évolution du Format Iceberg 511
Standardisation de l'Écosystème 511
Architecture Agentique et Lakehouse 512
Dynamiques de Marché et Consolidation 512
Évolution du Paysage Concurrentiel 512
Impact sur les Décisions d'Entreprise 513
Contexte Canadien 513
Recommandations Stratégiques 513
Pour les Organisations en Phase d'Évaluation 513
Pour les Organisations en Phase de Mise à l'Échelle 514
Pour les Organisations Matures 514
Feuille de Route d'Adoption 515
Phase 1 : Fondations (0-6 mois) 515
Phase 2 : Expansion (6-18 mois) 515
Phase 3 : Optimisation (18-36 mois) 516
Phase 4 : Innovation (36+ mois) 516
Études de Cas Prospectives 517
Cas 1 : Institution Financière Canadienne 517
Cas 2 : Détaillant Omnicanal 518
Cas 3 : Organisme Gouvernemental 518
Vision pour l'Avenir du Lakehouse d'Entreprise 519
Le Lakehouse Autonome 519
Le Lakehouse Conversationnel 519
Le Lakehouse Fédéré 519
Le Lakehouse Agentique 520
Résumé 520
Mot de la Fin 521
Annexe A - La Spécification Apache Iceberg 522
Introduction 522
A.1 Comprendre la spécification Iceberg 522
A.1.1 Philosophie et principes fondamentaux 522
A.1.2 Architecture en couches 523
A.1.3 Structure des métadonnées 523
A.1.4 Garanties transactionnelles 524
A.2 Versions du format de table Iceberg 524
A.2.1 Version 1 : Les fondations 524
A.2.2 Version 2 : Suppressions au niveau des lignes 524
A.2.3 Version 3 : Performance et expressivité 525
A.2.4 Compatibilité entre versions 526
A.3 Gestion des snapshots et métadonnées 526
A.3.1 Anatomie d'un snapshot 526
A.3.2 Structure des manifestes 526
A.3.3 Opérations sur les snapshots 527
A.3.4 Voyage dans le temps 527
A.3.5 Expiration des snapshots 527

A.3.6 Rollback 528
A.4 La spécification REST Catalog 528
A.4.1 Motivation et objectifs 528
A.4.2 Structure de l'API 528
A.4.3 Authentification et sécurité 529
A.4.4 Configuration et credentials de stockage 529
A.4.5 Implémentations notables 529
A.4.6 Évolutions futures 530
A.5 Spécification du format de fichier Puffin 530
A.5.1 Objectif et cas d'usage 530
A.5.2 Structure du fichier 530
A.5.3 Métadonnées de fichier 531
A.5.4 Types de blobs définis 531
A.5.5 Compression 531
A.5.6 Intégration avec Iceberg 531
A.5.7 Considérations de performance 532
A.6 Compatibilité et migration 532
A.6.1 Stratégies de migration vers Iceberg 532
A.6.2 Mise à niveau des versions de format 533
A.6.3 Compatibilité des moteurs 533
A.6.4 Considérations de rétrocompatibilité 533
A.6.5 Rollback de migration 534
A.6.6 Meilleures pratiques de migration 534
Résumé 534
Annexe B - Glossaire 536
Introduction 536
B.1 Terminologie Apache Iceberg 536
Catalog (Catalogue) 536
Compaction 536
Copy-on-Write (CoW) 536
Data File (Fichier de données) 537
Delete File (Fichier de suppression) 537
Deletion Vector (Vecteur de suppression) 537
Equality Delete (Suppression par égalité) 537
Hidden Partitioning (Partitionnement masqué) 537
Manifest File (Fichier de manifeste) 537
Manifest List (Liste de manifestes) 538
Merge-on-Read (MoR) 538
Metadata File (Fichier de métadonnées) 538
Partition Evolution (Évolution de partition) 538
Partition Pruning (Élagage de partition) 538
Position Delete (Suppression par position) 538
Puffin 539
REST Catalog 539
Row Lineage (Lignage des lignes) 539
Schema Evolution (Évolution de schéma) 539
Sequence Number (Numéro de séquence) 539
Snapshot 539
Snapshot Expiration (Expiration des snapshots) 540

Snapshot Isolation (Isolation des snapshots) 540
Statistics (Statistiques) 540
Time Travel (Voyage dans le temps) 540
Variant 540
B.2 Terminologie Lakehouse et Data Engineering 540
ACID 540
CDC (Change Data Capture) 541
Column Pruning (Élagage de colonnes) 541
Data Lake 541
Data Lakehouse 541
Data Warehouse 541
ELT (Extract, Load, Transform) 541
ETL (Extract, Transform, Load) 541
File Format (Format de fichier) 542
File Pruning (Élagage de fichiers) 542
Infonuagique (Cloud Computing) 542
Medallion Architecture (Architecture Médaillon) 542
Object Storage (Stockage objet) 542
OLAP (Online Analytical Processing) 542
Optimistic Concurrency (Concurrence optimiste) 542
Parquet 543
Query Engine (Moteur de requête) 543
Schema-on-Read 543
Schema-on-Write 543
Table Format (Format de table) 543
Write Amplification (Amplification d'écriture) 543
B.3 Terminologie Streaming et Kafka 543
Apache Kafka 543
Batch Processing (Traitement par lots) 544
Broker 544
Consumer 544
Consumer Group (Groupe de consommateurs) 544
Event Sourcing 544
Exactly-Once Semantics (Sémantiques exactly-once) 544
Kafka Connect 544
Lambda Architecture 545
Kappa Architecture 545
Offset 545
Partition (Kafka) 545
Producer 545
Sink Connector 545
Source Connector 545
Stream Processing (Traitement en flux) 545
Streaming Lakehouse 546
Topic 546
Watermark 546
B.4 Acronymes et abréviations 547
Index des termes par thème 549
Architecture et stockage 549

Apache Iceberg - Core 549
Apache Iceberg - Opérations 549
Apache Iceberg - Avancé 549
Performance et optimisation 549
Streaming et Kafka 549
Intégration de données 549
Références croisées vers les chapitres 550
Résumé 550

Chapitre IV.1 — Le Monde du Lakehouse Apache Iceberg

Introduction : L'Émergence d'un Nouveau Paradigme

L'architecture des données d'entreprise traverse actuellement une transformation sans précédent. Pendant des décennies, les organisations ont jonglé entre deux paradigmes fondamentaux : l'entrepôt de données (data warehouse) pour l'analytique structurée et le lac de données (data lake) pour le stockage massif et flexible. Cette dichotomie a engendré des infrastructures fragmentées, des coûts exponentiels et une complexité opérationnelle que peu d'équipes maîtrisaient véritablement.

En 2025, nous assistons à la consolidation d'une troisième voie : le **Data Lakehouse**. Cette architecture hybride promet de réconcilier la rigueur transactionnelle des entrepôts avec l'élasticité économique des lacs de données. Au cœur de cette révolution se trouve **Apache Iceberg**, un format de table ouvert qui s'impose comme le standard de facto pour les plateformes de données modernes.

Ce premier chapitre du Volume IV pose les fondations conceptuelles essentielles à la compréhension du Data Lakehouse. Nous explorerons l'évolution historique qui a mené à l'émergence d'Iceberg, les limitations des architectures précédentes qui ont motivé sa création, et les principes fondamentaux qui gouvernent cette nouvelle approche. Plus qu'un simple format de fichier, Apache Iceberg représente un changement de paradigme dans la façon dont les entreprises conçoivent, déploient et opèrent leurs plateformes de données.

Pour les architectes de données et les ingénieurs de plateforme canadiens, cette transformation revêt une importance particulière. Les exigences réglementaires strictes, notamment en matière de protection des renseignements personnels avec la Loi 25 au Québec et les lois provinciales sur la vie privée, nécessitent des architectures qui garantissent la traçabilité, la gouvernance et la conformité. Le Lakehouse Apache Iceberg offre précisément ces garanties, tout en maintenant la flexibilité requise pour l'innovation.

De l'Entrepôt au Lac : Une Évolution Nécessaire

Le Contexte Historique

Pour apprécier pleinement l'importance du Data Lakehouse, il convient de comprendre l'évolution historique des architectures de données d'entreprise. Cette évolution, s'étalant sur quatre décennies, révèle une progression vers des systèmes toujours plus flexibles et économiques, culminant avec le paradigme Lakehouse que nous explorons dans ce volume.

Dans les années 1980 et 1990, les bases de données relationnelles dominaient le paysage. Des systèmes comme Oracle, DB2 et Sybase servaient à la fois les besoins transactionnels et analytiques des entre-

prises. Cependant, à mesure que les volumes de données croissaient et que les besoins analytiques se complexifiaient, les limites de cette approche unifiée devinrent évidentes. Les requêtes analytiques lourdes perturbaient les opérations transactionnelles, et vice versa.

La réponse fut l'émergence de l'entrepôt de données (data warehouse) comme système dédié à l'analytique, séparé des systèmes opérationnels. Bill Inmon et Ralph Kimball, pionniers du domaine, établirent les fondations conceptuelles qui guident encore aujourd'hui de nombreuses implémentations. L'architecture typique impliquait l'extraction des données des systèmes sources, leur transformation pour conformité aux modèles analytiques, puis leur chargement dans l'entrepôt — le célèbre processus ETL (Extract, Transform, Load).

Les Limites de l'Entrepôt Traditionnel

L'entrepôt de données a dominé l'analytique d'entreprise depuis les années 1990. Des solutions comme Teradata, Oracle, puis Snowflake et Google BigQuery ont permis aux organisations de structurer leurs données pour l'intelligence d'affaires et la prise de décision. Ces systèmes excellaient dans leur domaine : requêtes SQL performantes, transactions ACID garanties, schémas rigoureux et gouvernance intégrée.

Cependant, l'explosion du volume de données et la diversification des cas d'usage ont révélé des limitations structurelles. Les entrepôts traditionnels imposent un modèle économique où le stockage et le calcul sont intimement couplés. Cette architecture monolithique génère des coûts qui croissent de façon non linéaire avec le volume de données. Pour une entreprise canadienne gérant des pétaoctets de données transactionnelles, de journaux applicatifs et de données IoT, les factures mensuelles peuvent atteindre des montants prohibitifs.

La rigidité schématique constitue une autre contrainte majeure. Dans un entrepôt classique, toute modification de schéma requiert une planification minutieuse, des fenêtres de maintenance et parfois des migrations coûteuses. Cette inflexibilité s'oppose directement aux pratiques agiles et aux besoins d'expérimentation rapide qui caractérisent les équipes de science des données modernes.

L'Avènement et les Promesses du Lac de Données

Face à ces limitations, le concept de lac de données (data lake) a émergé au début des années 2010. L'idée fondatrice était séduisante : stocker toutes les données brutes dans un système de fichiers distribué comme HDFS ou, plus tard, dans le stockage objet infonuagique (Amazon S3, Azure Blob Storage, Google Cloud Storage). Le traitement serait découplé du stockage, permettant une élasticité économique sans précédent.

Apache Hadoop, puis Apache Spark, ont fourni les moteurs de calcul nécessaires. Les organisations pouvaient désormais ingérer des données structurées, semi-structurées et non structurées sans transformation préalable. Le principe du « schema-on-read » libérait les équipes de la planification schématique rigide. Les coûts de stockage, particulièrement dans l'infonuagique, devenaient une fraction de ceux des entrepôts traditionnels.

Mais cette promesse s'est rapidement heurtée à la réalité opérationnelle. Sans les garanties transactionnelles des bases de données, les lacs de données se sont transformés en « marécages de données » (data swamps). L'absence de transactions atomiques signifiait que les écritures concurrentes pouvaient corrompre les données. Les lectures pendant les écritures retournaient des résultats incohérents. La gestion des mises à jour et des suppressions devenait un cauchemar opérationnel.

> **Étude de cas : Le Marécage de Données d'une Institution Financière Canadienne** *Secteur* : Services financiers — Banque de détail *Défi* : Une grande institution bancaire canadienne avait investi massivement

dans un lac de données Hadoop pour centraliser ses données client. Après trois ans d'exploitation, le lac contenait plus de 500 téraoctets de données, mais moins de 15 % étaient effectivement utilisables pour l'analytique. *Problèmes identifiés* : Données en double provenant de multiples ingestions non coordonnées, schémas incohérents entre les différents lots d'ingestion, impossibilité de mettre à jour les enregistrements existants (corrections de données client), absence de traçabilité sur l'origine et les transformations des données. *Impact* : Les équipes d'analytique passaient plus de 60 % de leur temps à nettoyer et préparer les données plutôt qu'à générer des insights. Les projets d'intelligence artificielle étaient constamment retardés par des problèmes de qualité de données. *Leçon* : Ce cas illustre parfaitement pourquoi le lac de données seul, sans les garanties d'un format de table comme Iceberg, ne peut répondre aux besoins analytiques de l'entreprise moderne.

Apache Hive : Une Première Tentative de Structure

Apache Hive a représenté la première tentative sérieuse d'apporter une structure de type entrepôt aux lacs de données. En introduisant un metastore centralisé et une interface SQL, Hive permettait aux analystes d'interroger les données du lac avec une syntaxe familière. Le système de partitionnement organisait les données en répertoires hiérarchiques, optimisant les requêtes qui filtraient sur les colonnes de partition.

Netflix, l'un des plus grands utilisateurs de Hive, a rapidement identifié les limitations fondamentales de cette approche. Avec un entrepôt de données dépassant les 100 pétaoctets stockés sur Amazon S3, l'équipe d'ingénierie de Netflix faisait face à des défis croissants. La découverte des fichiers nécessitait des listings de répertoires coûteux sur le stockage objet. Le metastore Hive devenait un goulot d'étranglement central. Plus critique encore, Hive ne pouvait garantir la cohérence des données lors d'écritures concurrentes.

Ryan Blue et Dan Weeks, ingénieurs chez Netflix, ont documenté ces problèmes avec précision. Les services et moteurs multiples qui accédaient aux tables Hive ne pouvaient compter sur la correction des données. L'absence de transactions atomiques stables rendait toute automatisation de maintenance risquée. Les équipes évitaient de modifier les données par crainte des conséquences imprévues. Cette situation intenable a conduit Netflix à concevoir une solution radicalement nouvelle : Apache Iceberg.

La Genèse d'Apache Iceberg

Netflix et la Naissance d'un Standard

En 2017, confrontée aux limitations de Hive à l'échelle de l'exaoctet, l'équipe de Netflix a entrepris de repenser fondamentalement la façon dont les tables analytiques devaient être structurées sur le stockage objet. L'objectif était ambitieux : créer un format de table qui apporterait les garanties d'un système de base de données moderne tout en préservant les avantages économiques du stockage infonuagique.

Le contexte de Netflix mérite qu'on s'y attarde. À cette époque, Netflix gérait l'une des plus grandes infrastructures de données au monde, servant plus de 200 millions d'abonnés avec des recommandations personnalisées alimentées par l'analyse de milliards d'événements quotidiens. Chaque lecture d'un utilisateur, chaque pause, chaque recherche générait des données qui alimentaient des modèles d'apprentissage automatique sophistiqués. L'entrepôt de données de Netflix avait crû pour dépasser les 100 pétaoctets stockés sur Amazon S3, avec des milliers de tables et des centaines de milliers de partitions.

Ryan Blue, architecte principal chez Netflix, a décrit le problème central en ces termes : sans commits atomiques, chaque modification d'une table Hive risquait de créer des erreurs de correction ailleurs.

L'automatisation de la maintenance était un rêve impossible, et les ingénieurs de données passaient un temps disproportionné à résoudre manuellement des problèmes de cohérence. Iceberg a été conçu spécifiquement pour permettre l'automatisation, même sur les systèmes de stockage objet infonuagique.

Les principes directeurs établis par les créateurs d'Iceberg reflétaient les leçons apprises de l'exploitation à grande échelle. La correction des données devait être garantie par des transactions ACID véritables, pas par des approximations ou des conventions. Les performances devaient s'améliorer grâce à des opérations à granularité fine au niveau des fichiers, éliminant les balayages de répertoires coûteux. L'exploitation et la maintenance des tables devaient être simplifiées et abstraites des détails d'implémentation.

Reconnaissant que cette innovation résolvait un problème universel, Netflix a fait don du projet à l'Apache Software Foundation en 2018. Cette décision stratégique a permis une gouvernance neutre et ouverte, attirant des contributions de l'ensemble de l'industrie. En 2020, Apache Iceberg est devenu un projet de premier niveau (top-level project) de l'Apache Foundation, confirmant sa maturité et son adoption croissante.

> **Migration : De Hive à Iceberg chez Netflix** *De* : Tables Hive sur HDFS puis S3, avec Hive Metastore centralisé *Vers* : Tables Iceberg avec REST Catalog développé en interne *Stratégie* : Migration progressive table par table, priorisant les tables les plus critiques et les plus volumineuses. Développement d'outils de migration automatisés permettant de convertir les métadonnées Hive en métadonnées Iceberg sans déplacement de données. Période de fonctionnement en parallèle pour validation avant bascule définitive. *Résultats* : Réduction de 80 % du temps de planification des requêtes pour les grandes tables, élimination complète des corruptions de données dues aux écritures concurrentes, automatisation de la maintenance des tables précédemment impossible.

Le Concept de Format de Table Ouvert

Pour comprendre l'innovation qu'apporte Iceberg, il faut distinguer trois niveaux dans l'architecture des données : le format de fichier, le format de table et le catalogue.

Les **formats de fichier** comme Apache Parquet, ORC ou Avro définissent comment les données sont physiquement encodées sur le disque. Parquet, par exemple, utilise un stockage colonnaire compressé optimisé pour les requêtes analytiques. Ces formats existaient bien avant Iceberg et continuent d'être utilisés comme couche de stockage sous-jacente.

Le **format de table** constitue l'innovation centrale d'Iceberg. Il définit une spécification ouverte pour organiser une collection de fichiers de données en une table logique cohérente. Cette couche de méta-données ajoute les fonctionnalités de base de données : transactions atomiques, évolution de schéma, partitionnement intelligent et historique des versions. Le format de table transforme un ensemble de fichiers Parquet dispersés en une structure interrogeable avec des garanties de cohérence.

Le **catalogue** représente la couche supérieure qui maintient le registre des tables et leur localisation. Différentes implémentations de catalogue existent : Hive Metastore, AWS Glue, Apache Polaris ou le REST Catalog standard d'Iceberg. Le catalogue pointe vers le fichier de métadonnées courant de chaque table, servant de point d'entrée pour tous les moteurs de requête.

Positionnement dans l'Écosystème

Apache Iceberg n'est pas seul dans l'espace des formats de table ouverts. Delta Lake, développé par Databricks, et Apache Hudi, créé chez Uber, offrent des fonctionnalités similaires avec des approches architecturales distinctes.

Delta Lake a émergé vers 2017-2018 chez Databricks pour apporter des transactions fiables au stockage objet infonuagique. Son architecture repose sur un journal de transactions (_delta_log) qui enregistre séquentiellement chaque modification de la table. L'intégration profonde avec Apache Spark et l'écosystème Databricks en fait un choix naturel pour les organisations déjà investies dans cette plateforme.

Apache Hudi, développé initialement chez Uber, se distingue par son orientation vers les mises à jour incrémentales et la capture de changements de données (CDC). Son architecture offre deux modes de stockage — Copy-on-Write et Merge-on-Read — permettant aux utilisateurs d'optimiser soit les performances de lecture, soit celles d'écriture selon leurs besoins.

La comparaison technique entre ces trois formats révèle des philosophies différentes :

Critère	Apache Iceberg	Delta Lake	Apache Hudi
Gouvernance	Apache Software Foundation	Linux Foundation (Databricks)	Apache Software Foundation
Support multi-moteur	Excellent (natif)	Bon (amélioration récente)	Bon
Évolution de schéma	Complète sans réécriture	Supportée	Supportée avec limitations
Partitionnement masqué	Natif	Non disponible	Non disponible
Évolution de partition	Sans réécriture	Requiert réécriture	Requiert réécriture
Performances d'écriture	Excellentes	Excellentes	Optimisées pour CDC
Maturité streaming	Bonne	Excellente (Spark)	Excellente (natif)
Complexité opérationnelle	Modérée	Faible (Databricks)	Élevée

En 2025, le paysage a considérablement évolué. Apache Iceberg s'est imposé comme le standard de facto, bénéficiant du support natif des principaux fournisseurs infonuagiques (AWS, Google Cloud, Azure), des plateformes de données (Snowflake, Databricks) et des moteurs de requête (Spark, Trino, Flink). Cette convergence vers Iceberg reflète sa conception orientée vers l'interopérabilité et son statut de projet véritablement ouvert sous gouvernance Apache.

L'émergence d'Apache XTable (anciennement OneTable) témoigne de la maturité de l'écosystème. Ce projet, co-lancé par Microsoft, Google et Onehouse, permet l'interopérabilité entre les formats de table, éliminant la nécessité de choisir un format unique. Une table peut être exposée simultanément comme Iceberg, Delta et Hudi, chaque moteur accédant aux données via son format préféré.

Le Paradigme du Data Lakehouse

Définition et Principes Fondateurs

Le terme « Lakehouse » désigne une architecture qui combine les meilleurs attributs des entrepôts de données et des lacs de données. Cette fusion n'est pas un simple compromis, mais une synthèse architecturale qui élimine les limitations de chaque approche précédente.

Du lac de données, le Lakehouse hérite le stockage économique et élastique sur des systèmes de fichiers distribués ou le stockage objet infonuagique. Les données restent dans des formats ouverts comme Parquet, accessibles par n'importe quel outil compatible. Le découplage du stockage et du calcul permet une élasticité économique remarquable : le stockage coûte une fraction des solutions intégrées, et les ressources de calcul s'ajustent dynamiquement aux besoins.

De l'entrepôt de données, le Lakehouse adopte les garanties transactionnelles ACID, la gouvernance des données, l'évolution de schéma contrôlée et les optimisations de requêtes. Les utilisateurs peuvent exécuter des requêtes SQL avec des performances comparables aux entrepôts traditionnels, tout en bénéficiant de la flexibilité du lac de données sous-jacent.

Apache Iceberg fournit la couche technique qui rend cette synthèse possible. Son système de métadonnées hiérarchique transforme une collection de fichiers Parquet en une table avec des propriétés de base de données. Les commits atomiques garantissent que les lecteurs voient toujours un état cohérent de la table. L'évolution de schéma permet des modifications sans réécriture des données existantes. Le partitionnement masqué (hidden partitioning) optimise automatiquement les requêtes sans exposer les détails de partitionnement aux utilisateurs.

Les Cinq Couches de l'Architecture Lakehouse

Une architecture Lakehouse moderne se structure typiquement en cinq couches distinctes, chacune avec des responsabilités claires.

La **couche de stockage** constitue la fondation. Elle repose généralement sur le stockage objet infonuagique (S3, Azure Blob, GCS) ou des systèmes de fichiers distribués. Cette couche fournit durabilité, disponibilité et économie d'échelle. Les fichiers de données et les fichiers de métadonnées d'Iceberg résident tous deux dans cette couche.

La **couche de format de table** est où Apache Iceberg opère. Elle définit comment les fichiers de données sont organisés en tables logiques, comment les transactions sont gérées et comment les métadonnées évoluent. Cette couche est cruciale pour les garanties de cohérence et les optimisations de performance.

La **couche d'ingestion** gère le flux de données entrant dans le Lakehouse. Elle peut inclure des pipelines de traitement par lots avec Spark, des flux en temps réel avec Apache Flink ou Apache Kafka, ou des connecteurs CDC pour capturer les changements des systèmes sources. L'intégration entre Kafka et Iceberg, que nous explorerons en détail dans les chapitres subséquents, forme ce qu'on appelle le Streaming Lakehouse.

La **couche de catalogue** maintient le registre des tables et orchestre l'accès concurrent. Des solutions comme Apache Polaris, AWS Glue Catalog ou le REST Catalog d'Iceberg fournissent cette fonctionnalité. Le catalogue est le point d'entrée pour la découverte des données et la gestion des permissions.

La **couche de consommation** expose les données aux utilisateurs finaux et aux applications. Elle inclut les moteurs de requête SQL (Trino, Spark SQL, Dremio), les outils de visualisation (Power BI, Tableau) et

les frameworks d'apprentissage automatique. La force d'Iceberg réside dans sa capacité à servir simultanément ces différents consommateurs sur les mêmes données.

Avantages Stratégiques pour l'Entreprise

L'adoption d'une architecture Lakehouse basée sur Apache Iceberg procure des avantages stratégiques mesurables pour les organisations.

La **réduction des coûts** est souvent le bénéfice le plus immédiatement quantifiable. Le stockage objet infonuagique coûte typiquement une fraction du stockage dans les entrepôts de données traditionnels. Le découplage calcul-stockage permet d'ajuster les ressources de calcul indépendamment, évitant le surprovisionnement chronique des solutions intégrées. Environ la moitié des organisations rapportent des économies significatives après l'adoption d'une architecture Lakehouse.

> **Performance : Économies de Coûts Documentées** Les organisations qui migrent vers une architecture Lakehouse Iceberg rapportent typiquement : - Réduction de 50 à 70 % des coûts de stockage par rapport aux entrepôts propriétaires - Réduction de 30 à 50 % des coûts de calcul grâce à l'élasticité - Élimination des coûts de licence des solutions propriétaires - Réduction des coûts opérationnels par l'automatisation de la maintenance

L'**élimination du verrouillage fournisseur** constitue un avantage stratégique majeur. Avec Iceberg, les données restent dans des formats ouverts accessibles par n'importe quel moteur compatible. Une organisation peut utiliser Spark pour l'ingestion, Trino pour les requêtes interactives et Flink pour le traitement en temps réel, tous sur les mêmes tables. Cette flexibilité permet de choisir les meilleurs outils pour chaque cas d'usage et facilite les migrations futures.

L'**unification des charges de travail** simplifie l'architecture globale. Plutôt que de maintenir des copies séparées des données pour l'analytique, la science des données et l'apprentissage automatique, le Lakehouse fournit une source de vérité unique. Cette consolidation réduit les incohérences entre systèmes, simplifie la gouvernance et accélère la mise en œuvre de nouveaux cas d'usage.

La **gouvernance native** répond aux exigences réglementaires croissantes. Iceberg supporte le contrôle d'accès au niveau des lignes et des colonnes, l'historique complet des modifications via les snapshots, et les capacités de voyage dans le temps (time travel) essentielles pour l'audit. Pour les organisations canadiennes soumises à la LPRPDE fédérale et aux lois provinciales sur la protection de la vie privée, ces fonctionnalités facilitent la conformité.

L'**agilité analytique** accélère le temps de mise en valeur des données. Les équipes de science des données peuvent expérimenter sur les données de production sans créer de copies, grâce à l'isolation des snapshots. Les modifications de schéma s'effectuent sans interruption de service. Les nouvelles sources de données s'intègrent rapidement sans planification schématique préalable exhaustive.

Apache Iceberg : Vue d'Ensemble Technique

Architecture à Trois Niveaux

L'architecture d'Apache Iceberg se distingue par sa structure hiérarchique de métadonnées soigneusement conçue. Cette organisation en niveaux permet des opérations efficaces à grande échelle tout en maintenant des garanties de cohérence forte.

Au niveau supérieur, le **catalogue** maintient un pointeur vers le fichier de métadonnées courant de chaque table. Ce pointeur constitue le seul élément qui change lors d'un commit : la mise à jour atomique de cette référence est ce qui garantit les transactions ACID. Différentes implémentations de catalogue existent pour différents environnements — Hive Metastore pour les déploiements Hadoop existants, AWS Glue pour les environnements AWS natifs, REST Catalog pour l'interopérabilité multi-moteur.

La **couche de métadonnées** (metadata layer) comprend trois composants hiérarchiques. Le fichier de métadonnées principal (metadata.json) contient la description complète de la table : schéma, spécifications de partition, liste des snapshots et propriétés de configuration. Chaque snapshot pointe vers une liste de manifestes (manifest list) qui indexe les fichiers manifestes associés à cet état de la table. Les fichiers manifestes (manifest files) contiennent l'inventaire détaillé des fichiers de données, incluant leurs statistiques au niveau des colonnes.

La **couche de données** (data layer) contient les fichiers de données réels, typiquement au format Parquet pour les charges analytiques. Iceberg supporte également ORC et Avro selon les besoins. Ces fichiers sont immuables une fois écrits — toute modification crée de nouveaux fichiers plutôt que de modifier les existants.

Le Rôle Central des Métadonnées

La philosophie d'Iceberg peut se résumer ainsi : les métadonnées sont l'architecture. Contrairement aux approches précédentes où les métadonnées étaient une réflexion tardive, Iceberg place le système de métadonnées au cœur de sa conception.

Le **fichier de métadonnées** (metadata file) sert de description autoritaire de la table à un instant donné. Il contient la version du format Iceberg, l'identifiant unique de la table (UUID), l'emplacement de base pour les fichiers de données, l'historique des schémas, les spécifications de partition et la liste des snapshots valides. Lorsqu'une table évolue, un nouveau fichier de métadonnées est créé, préservant l'historique complet.

Chaque **snapshot** représente l'état complet de la table à un moment précis. Un snapshot n'est pas un delta ou une différence, mais une vue complète qui peut être interrogée indépendamment. Cette approche simplifie considérablement les requêtes de voyage dans le temps : pour lire la table telle qu'elle était hier, il suffit de naviguer vers le snapshot correspondant.

Le **voyage dans le temps** (time travel) constitue l'une des fonctionnalités les plus puissantes d'Iceberg pour les cas d'usage d'entreprise. Cette capacité permet de :

- **Reproduire les analyses** : Les scientifiques de données peuvent réexécuter des analyses sur l'état exact des données à un moment donné, garantissant la reproductibilité des résultats.
- **Déboguer les problèmes de données** : Lorsqu'une anomalie est détectée, les équipes peuvent examiner les états précédents pour identifier quand et comment le problème est apparu.
- **Supporter l'audit** : Les auditeurs peuvent vérifier l'état des données à n'importe quelle date passée, essentiel pour la conformité réglementaire.

- **Récupérer des erreurs** : Si une erreur de traitement corrompt des données, il est possible de revenir à un snapshot antérieur sans restauration de sauvegarde complète.

La syntaxe de voyage dans le temps varie selon le moteur de requête, mais le principe reste identique. Par exemple, avec Spark SQL :

```
-- Lecture de la table à un timestamp spécifique
SELECT * FROM catalog.db.table
TIMESTAMP AS OF '2025-01-15 10:00:00'

-- Lecture de la table à un snapshot spécifique
SELECT * FROM catalog.db.table
VERSION AS OF 12345678901234567890
```

La rétention des snapshots est configurable. Par défaut, Iceberg conserve les snapshots indéfiniment, mais les organisations configurent typiquement une politique d'expiration (par exemple, 30 jours) pour contrôler les coûts de stockage tout en maintenant un historique suffisant pour leurs besoins opérationnels et réglementaires.

La **liste de manifestes** (manifest list) associée à chaque snapshot indexe les fichiers manifestes qui composent cet état. Elle contient également des statistiques agrégées sur les valeurs de partition dans chaque manifeste. Cette information permet l'élagage au niveau du manifeste (manifest-level pruning) : les moteurs de requête peuvent éliminer des manifestes entiers sans les lire si leurs plages de partition ne correspondent pas aux filtres de la requête.

Les **fichiers manifestes** (manifest files) contiennent l'inventaire détaillé des fichiers de données. Pour chaque fichier, le manifeste enregistre l'emplacement, la taille, le nombre de lignes, les valeurs de partition et des statistiques au niveau des colonnes (minimum, maximum, nombre de valeurs nulles). Ces statistiques permettent l'élagage au niveau des fichiers (file-level pruning), évitant la lecture de fichiers qui ne peuvent contenir de données pertinentes pour la requête.

Transactions ACID et Isolation

L'un des problèmes fondamentaux que Hive ne pouvait résoudre était la cohérence lors d'écritures concurrentes. Iceberg implémente un véritable support transactionnel ACID qui garantit la correction des données même sous charge intensive.

L'**atomicité** est garantie par le mécanisme de commit basé sur le pointeur de métadonnées. Un écrivain prépare tous les nouveaux fichiers de données, génère les manifestes correspondants et crée un nouveau fichier de métadonnées. Ces artefacts sont invisibles jusqu'à ce que le pointeur du catalogue soit atomiquement mis à jour. Si une opération échoue avant le commit, les fichiers temporaires sont simplement nettoyés — la table reste dans son état précédent.

Le processus de commit suit une séquence précise : 1. L'écrivain lit l'état courant de la table depuis le catalogue 2. Les fichiers de données sont écrits dans le stockage objet 3. De nouveaux fichiers manifestes sont créés, référençant les fichiers de données 4. Une nouvelle liste de manifestes est générée, pointant vers les manifestes mis à jour 5. Un nouveau fichier de métadonnées est écrit, référençant la nouvelle liste de manifestes 6. Le pointeur du catalogue est atomiquement mis à jour vers le nouveau fichier de métadonnées

La **cohérence** est maintenue par le système de snapshots. Chaque lecteur voit un état cohérent de la table : soit l'état avant une transaction, soit l'état après, jamais un état intermédiaire. Les lecteurs peuvent même choisir explicitement quel snapshot consulter pour des besoins de reproductibilité.

L'**isolation** est fournie par le modèle d'isolation snapshot. Les lecteurs travaillent sur un snapshot spécifique et ne sont pas affectés par les écritures concurrentes. Les écrivains utilisent un contrôle de concurrence optimiste : ils préparent leurs modifications basées sur l'état courant, puis tentent un commit atomique. Si l'état a changé entre-temps (un autre écrivain a committé), le commit échoue et peut être réessayé.

Le contrôle de concurrence optimiste fonctionne ainsi : 1. L'écrivain note l'identifiant du snapshot courant au début de l'opération 2. L'écrivain prépare ses modifications 3. Lors du commit, l'écrivain vérifie que le snapshot courant n'a pas changé 4. Si le snapshot a changé, le commit échoue avec un conflit de concurrence 5. L'écrivain peut alors réessayer l'opération avec le nouvel état

Ce modèle diffère du verrouillage pessimiste utilisé par les bases de données traditionnelles. Il est particulièrement adapté aux charges analytiques où les conflits sont rares mais les transactions peuvent être longues. Pour les charges avec beaucoup de contention, des stratégies de partitionnement ou de séquencement des écritures peuvent être nécessaires.

La **durabilité** est héritée du système de stockage sous-jacent. Une fois qu'un commit réussit et que le pointeur de métadonnées est mis à jour, les modifications sont permanentes et survivent aux pannes du système.

> **Migration : Garanties Transactionnelles** *De* : Lac de données sans garanties transactionnelles (écritures Parquet directes sur S3) *Vers* : Tables Iceberg avec ACID complet *Stratégie* : Validation des propriétés ACID avant migration de production. Tests de charge avec écritures concurrentes pour vérifier l'absence de corruption. Monitoring des conflits de concurrence optimiste pour ajuster les stratégies de partitionnement. *Points d'attention* : Les applications existantes qui assumaient un modèle eventually consistent peuvent nécessiter des ajustements pour tirer parti des garanties ACID.

Évolution de Schéma

La capacité d'Iceberg à supporter l'évolution de schéma (schema evolution) sans réécriture de données représente une avancée majeure par rapport aux systèmes précédents.

Les opérations de schéma supportées incluent l'ajout de nouvelles colonnes, la suppression de colonnes existantes, le renommage de colonnes et la modification de types (dans certaines limites compatibles). Ces modifications sont enregistrées dans les métadonnées de la table et prennent effet immédiatement pour les nouvelles écritures.

La clé de cette flexibilité réside dans le système d'identifiants de colonnes. Chaque colonne dans Iceberg est identifiée par un ID unique plutôt que par son nom ou sa position. Lorsqu'un fichier de données est lu, les colonnes sont mappées par ID vers le schéma courant de la table. Les colonnes ajoutées après la création d'un fichier retournent null. Les colonnes supprimées sont simplement ignorées. Les colonnes renommées continuent d'être correctement associées grâce à leur ID.

Cette approche contraste fortement avec Hive, où les colonnes étaient identifiées par position. Un simple renommage de colonne dans Hive pouvait corrompre l'interprétation des données existantes. Iceberg élimine cette classe entière de problèmes.

Partitionnement Masqué

Le partitionnement traditionnel dans Hive exposait les détails de partition aux utilisateurs. Les requêtes devaient explicitement filtrer sur les colonnes de partition dans leur syntaxe exacte pour bénéficier de

l'élagage de partition. Un filtrage sur une fonction de la colonne de partition (comme YEAR(date) ou HOUR(timestamp)) ne déclenchait pas l'élagage.

Iceberg introduit le concept de **partitionnement masqué** (hidden partitioning) qui abstrait ces détails. Les spécifications de partition dans Iceberg peuvent inclure des transformations : year(), month(), day(), hour(), bucket(), truncate(). Les moteurs de requête utilisent les métadonnées pour déterminer automatiquement quelles partitions éliminer, même si la requête utilise la colonne source plutôt que la transformation de partition.

Par exemple, une table partitionnée par month(event_time) permettra l'élagage de partition pour une requête filtrant sur event_time BETWEEN "2025-01-01" AND "2025-01-31". L'utilisateur n'a pas besoin de connaître la stratégie de partitionnement — Iceberg détermine automatiquement que seule la partition du mois de janvier 2025 est pertinente.

Cette abstraction a une conséquence importante : l'**évolution de partition** (partition evolution). Iceberg permet de modifier la stratégie de partitionnement sans réécrire les données existantes. Si une table initialement partitionnée par mois doit être partitionnée par jour pour améliorer la granularité, la nouvelle spécification s'applique aux futures écritures tandis que les données historiques conservent leur partitionnement original. Les requêtes fonctionnent transparentement sur les deux schémas de partition.

Stratégies d'Écriture : Copy-on-Write et Merge-on-Read

Apache Iceberg supporte deux stratégies fondamentales pour gérer les mises à jour et les suppressions de données : Copy-on-Write (CoW) et Merge-on-Read (MoR). Le choix entre ces stratégies impacte significativement les performances selon le profil de charge de travail.

Copy-on-Write réécrit les fichiers de données entiers lors des modifications. Lorsqu'une ligne est mise à jour ou supprimée, Iceberg identifie le fichier de données contenant cette ligne, lit le fichier, applique la modification et écrit un nouveau fichier complet. L'ancien fichier est marqué pour suppression dans les métadonnées.

Les avantages de CoW incluent : - Lectures très performantes car aucune réconciliation n'est nécessaire - Simplicité de mise en œuvre et de débogage - Métadonnées plus simples sans fichiers de suppression séparés

Les inconvénients de CoW : - Écritures coûteuses pour les mises à jour éparses sur de grands fichiers - Amplification d'écriture significative - Latence plus élevée pour les opérations de mise à jour

Merge-on-Read utilise des fichiers de suppression (delete files) pour enregistrer les modifications sans réécrire les données existantes. Lors d'une mise à jour, Iceberg écrit un fichier de suppression positionnelle indiquant quelles lignes sont supprimées, puis écrit un nouveau fichier contenant les lignes mises à jour. Lors de la lecture, le moteur de requête fusionne les fichiers de données avec les fichiers de suppression pour produire le résultat correct.

Les avantages de MoR incluent : - Écritures beaucoup plus rapides pour les mises à jour éparses - Faible amplification d'écriture - Meilleure latence pour les modifications fréquentes

Les inconvénients de MoR : - Lectures plus lentes en raison de la réconciliation nécessaire - Métadonnées plus complexes - Nécessité de compaction plus fréquente pour maintenir les performances de lecture

Le choix entre CoW et MoR dépend du profil de charge de travail. Pour les charges avec peu de mises à jour et beaucoup de lectures (typique de l'analytique), CoW est généralement préférable. Pour les charges avec des mises à jour fréquentes (CDC, IoT), MoR peut offrir de meilleures performances globales. Iceberg permet de configurer cette stratégie au niveau de la table et même de changer dynamiquement selon les besoins.

Écosystème et Adoption Industrielle

Support Multi-Moteur

La force distinctive d'Apache Iceberg réside dans son interopérabilité. Contrairement à Delta Lake qui reste fortement couplé à l'écosystème Databricks/Spark, Iceberg a été conçu dès l'origine pour fonctionner avec n'importe quel moteur de calcul.

Apache Spark fut l'un des premiers moteurs à intégrer Iceberg, naturellement puisque Iceberg est né de l'utilisation de Spark chez Netflix avec des ensembles de données de l'ordre du pétaoctet. L'intégration Spark-Iceberg permet les opérations DDL complètes, les requêtes MERGE INTO pour les upserts, et le support des flux structurés (Structured Streaming) pour l'ingestion en temps réel.

Trino (anciennement PrestoSQL) offre une intégration native d'Iceberg particulièrement adaptée aux requêtes interactives. Le connecteur Iceberg de Trino supporte l'élagage de partition, les statistiques de table et le voyage dans le temps. Pour les organisations qui utilisent Trino comme moteur de requête fédéré, Iceberg s'intègre naturellement dans l'architecture existante.

Apache Flink fournit l'intégration streaming essentielle pour le Streaming Lakehouse. Flink peut consommer des flux Kafka et écrire directement dans des tables Iceberg avec des garanties exactement-une-fois (exactly-once). Cette intégration sera explorée en profondeur dans le chapitre sur le Streaming Lakehouse, en lien avec les concepts développés dans le Volume III sur Apache Kafka.

Dremio se positionne comme une plateforme Lakehouse unifiée avec une intégration Iceberg de premier ordre. Son moteur de requête optimisé, combiné à des fonctionnalités comme la réflexion de données (data reflections) pour l'accélération de requêtes, en fait un choix populaire pour les déploiements Lakehouse de production.

Snowflake a ajouté le support des tables Iceberg, permettant aux clients d'interroger des données Iceberg stockées dans leur propre stockage objet tout en bénéficiant de l'optimiseur de requêtes Snowflake. Cette approche « BYOS » (Bring Your Own Storage) étend le modèle Snowflake vers l'interopérabilité ouverte.

Convergence de l'Industrie

L'année 2024 a marqué un tournant décisif dans l'adoption d'Iceberg, avec des annonces majeures des principaux acteurs de l'industrie.

L'acquisition de **Tabular** par Databricks a fait les manchettes. Tabular, fondée par les créateurs originaux d'Iceberg (Ryan Blue et Dan Weeks), développait un service de gestion de tables Iceberg. Cette acquisition signale l'engagement de Databricks envers l'interopérabilité, même si Delta Lake reste leur format natif.

Snowflake a annoncé l'open-sourcing de **Polaris Catalog**, son implémentation de catalogue Iceberg. Cette décision vise à fournir aux entreprises une solution de catalogue ouverte avec sécurité de niveau entreprise et interopérabilité complète avec l'écosystème Iceberg.

Microsoft a intégré Iceberg dans **Fabric** via OneLake, permettant aux organisations d'utiliser des tables Iceberg avec Power BI Direct Lake et l'ensemble des outils Fabric. Cette intégration, que nous explorerons en détail dans un chapitre dédié, positionne Iceberg comme pont entre les mondes open-source et Microsoft.

Les fournisseurs infonuagiques ont également renforcé leur support. **AWS** offre Iceberg nativement dans Athena, EMR et Glue. **Google Cloud** intègre Iceberg dans BigQuery, permettant l'interrogation de tables Iceberg externes. **Azure** supporte Iceberg via Synapse Analytics et HDInsight.

Adoption par les Géants Technologiques

L'adoption d'Iceberg par les entreprises technologiques de premier plan valide sa maturité pour les charges de travail de production critiques.

Netflix, créateur d'Iceberg, a complété sa migration vers une architecture exclusivement Iceberg. Leur entrepôt de données, qui dépasse l'exaoctet, utilise maintenant Iceberg pour toutes les tables. L'équipe a développé un outillage personnalisé, des services écosystémiques et des fonctionnalités uniques comme les tables Iceberg sécurisées et le REST Catalog Iceberg.

Apple utilise Iceberg comme fondation de son architecture Lakehouse à travers toutes ses divisions. Leurs tables vont de quelques centaines de mégaoctets à plusieurs pétaoctets, gérant des charges de travail de streaming en temps réel, de micro-lots et d'ETL traditionnel. L'équipe d'ingénierie d'Apple a développé des versions distribuées des procédures de maintenance Iceberg pour opérer à leur échelle.

Russell Spitzer, Engineering Manager chez Apple et membre du PMC Apache Iceberg, a partagé un défi particulier : la gestion des exigences de conformité réglementaire comme le RGPD et le Digital Markets Act (DMA) européen, qui exigent des opérations au niveau des lignes plutôt qu'au niveau des partitions. Les systèmes traditionnels comme Hive ne pouvaient mettre à jour que des partitions entières, rendant les opérations de conformité extrêmement coûteuses pour des mises à jour éparses sur des ensembles de données massifs. Apple a développé des capacités Copy-on-Write et Merge-on-Read pour permettre des opérations efficaces au niveau des lignes.

LinkedIn a migré vers Iceberg pour son infrastructure de données analytiques. Le réseau professionnel bénéficie particulièrement de l'évolution de schéma et du partitionnement masqué pour gérer l'évolution constante de ses modèles de données.

Airbnb utilise Iceberg pour ses analyses de données massives, rapportant des améliorations significatives en fiabilité et performance après migration depuis leur architecture précédente.

> **Étude de cas : Siemens — Lakehouse Industriel à l'Échelle Mondiale** *Secteur* : Fabrication industrielle et santé (Siemens Digital Industries et Siemens Healthineers) *Défi* : Siemens devait unifier les données de milliers d'équipements industriels et de dispositifs médicaux répartis mondialement, tout en supportant des analyses en temps réel pour la maintenance prédictive et l'optimisation de la production. *Solution* : Implémentation d'une architecture « Shift Left » combinant Apache Kafka pour l'ingestion événementielle et Apache Iceberg pour le stockage analytique. Les données sont validées, enrichies et gouvernées dès leur ingestion, avant l'écriture dans les tables Lakehouse. *Résultats* : Données de production disponibles pour l'analytique avec une latence inférieure à 5 minutes, réduction de 40 % des temps d'arrêt non planifiés grâce à la maintenance prédictive, conformité automatisée avec les réglementations industrielles internationales. *Leçon* : L'architecture Streaming Lakehouse (Kafka + Iceberg) représente l'avenir de l'analytique industrielle en temps réel.

Tendances d'Adoption par Secteur

L'adoption d'Apache Iceberg varie selon les secteurs, chacun présentant des motivations et des défis spécifiques.

Dans les **services financiers**, l'adoption est motivée par les exigences réglementaires strictes, le besoin de traçabilité complète et les volumes massifs de données transactionnelles. Les banques et les assureurs utilisent Iceberg pour unifier les données de risque, de conformité et d'analytique client. TD Bank, par exemple, a annoncé l'utilisation de Databricks (avec support Iceberg) sur Microsoft Azure pour moderniser ses capacités analytiques.

Le secteur de la **technologie et des médias** adopte Iceberg pour gérer les flux de données utilisateur à grande échelle. Les plateformes de streaming, les réseaux sociaux et les services de jeux en ligne bénéficient particulièrement du support du streaming en temps réel et de l'évolution de schéma flexible.

Dans le **commerce de détail et le e-commerce**, Iceberg permet l'unification des données de point de vente, de commerce électronique et de chaîne d'approvisionnement. La capacité de segmentation client en temps quasi réel et l'analytique de panier sont des cas d'usage courants.

Le secteur de la **santé** utilise Iceberg pour les analyses de données cliniques, la recherche et l'amélioration des opérations. Les fonctionnalités de contrôle d'accès granulaire sont essentielles pour la conformité aux réglementations de protection des données de santé.

Dans l'**énergie et les services publics**, les cas d'usage incluent l'analytique des compteurs intelligents, l'optimisation du réseau et la maintenance prédictive des équipements. Les volumes de données IoT massifs et les exigences de latence font d'Iceberg un choix naturel.

Perspectives Canadiennes et Contexte Réglementaire

Exigences Réglementaires Distinctives

Le contexte canadien présente des exigences réglementaires qui rendent l'adoption du Lakehouse Apache Iceberg particulièrement pertinente.

La **Loi sur la protection des renseignements personnels et les documents électroniques** (LPRPDE) au niveau fédéral, combinée aux lois provinciales comme la **Loi 25** au Québec, impose des obligations strictes en matière de protection des données personnelles. Les organisations doivent pouvoir démontrer la traçabilité des données, implémenter le droit à l'effacement (droit à l'oubli) et maintenir des journaux d'audit complets.

Les capacités d'Iceberg répondent directement à ces exigences. Le voyage dans le temps permet de reconstruire l'état des données à n'importe quel moment pour les audits. Les opérations de suppression au niveau des lignes (row-level deletes) facilitent la conformité au droit à l'effacement sans réécrire des partitions entières. Les métadonnées de lignage intégrées documentent l'origine et les transformations des données.

Le secteur des **services financiers** canadien, régulé par le Bureau du surintendant des institutions financières (BSIF), impose des exigences additionnelles en matière de gouvernance des données. Les institutions financières doivent maintenir des contrôles d'accès granulaires, des pistes d'audit complètes et des capacités de reprise après sinistre. L'architecture Lakehouse, avec son stockage sur le stockage objet infonuagique répliqué et ses métadonnées versionnées, fournit ces garanties nativement.

Le secteur de la **santé**, avec les lois provinciales sur la protection des renseignements de santé, nécessite une isolation stricte des données sensibles. Les fonctionnalités de contrôle d'accès au niveau des colonnes

d'Iceberg permettent de masquer ou restreindre l'accès aux informations médicales sensibles tout en permettant l'analytique sur les données désidentifiées.

Étude de cas : Transformation Lakehouse dans le Commerce de Détail Canadien *Secteur* : Commerce de détail — Chaîne nationale avec présence au Québec et en Ontario *Défi* : Un détaillant majeur canadien opérant plus de 500 magasins devait moderniser son infrastructure de données pour supporter ses initiatives d'analytique avancée et de personnalisation client. L'architecture existante, basée sur un entrepôt de données Oracle et des lacs de données S3 non structurés, créait des silos et des incohérences. La conformité à la Loi 25 du Québec, exigeant des capacités de suppression des données personnelles sur demande, représentait un défi majeur. *Solution* : Déploiement d'une architecture Lakehouse basée sur Apache Iceberg sur AWS (région Canada-Montréal) avec Trino comme moteur de requête principal. Migration progressive des données de l'entrepôt Oracle vers des tables Iceberg. Implémentation d'un pipeline de traitement des demandes de suppression utilisant les capacités de suppression au niveau des lignes d'Iceberg. *Architecture* : Tables Iceberg stockées sur S3 (région ca-central-1), AWS Glue comme catalogue Iceberg, Trino pour les requêtes interactives, Apache Spark pour les transformations par lots, Apache Kafka (intégré avec Iceberg via Flink) pour l'ingestion en temps réel des transactions de point de vente. *Résultats* : Réduction de 65 % des coûts d'infrastructure de données, traitement des demandes de suppression RGPD/Loi 25 en moins de 24 heures (vs 2 semaines auparavant), temps de génération des rapports hebdomadaires réduit de 4 heures à 15 minutes, capacité de segmentation client en temps quasi réel pour les promotions personnalisées. *Leçon* : La combinaison de la conformité réglementaire et de l'agilité analytique représente un différenciateur compétitif dans le contexte canadien.

Souveraineté des Données

La question de la souveraineté des données prend une importance croissante pour les organisations canadiennes. Plusieurs réglementations sectorielles exigent que certaines catégories de données demeurent sur le territoire canadien.

L'architecture Lakehouse facilite la conformité à ces exigences. Le stockage objet infonuagique peut être configuré pour garantir la résidence des données dans des régions spécifiques (par exemple, les régions AWS Canada à Montréal, Azure Canada à Toronto et Québec, ou Google Cloud à Montréal). Les métadonnées Iceberg étant stockées avec les données, aucune dépendance externe ne compromet la souveraineté.

Cette flexibilité permet également des architectures hybrides où certaines données sensibles restent sur site (dans des centres de données canadiens) tandis que d'autres utilisent l'infonuagique public. Iceberg fonctionne de manière identique sur les deux environnements, simplifiant l'architecture globale.

Les considérations de souveraineté incluent également :

Classification des données : Les organisations doivent classifier leurs données selon leur sensibilité et les exigences de résidence applicables. Iceberg supporte les propriétés de table personnalisées qui peuvent encoder cette classification et guider les politiques d'accès.

Réplication contrôlée : Pour les besoins de reprise après sinistre, les données doivent parfois être répliquées dans une région secondaire. Cette réplication doit respecter les contraintes de souveraineté — par exemple, les données soumises à résidence canadienne peuvent être répliquées entre Toronto et Montréal, mais pas vers une région américaine.

Accès transfrontalier : Même si les données résident au Canada, l'accès depuis d'autres juridictions peut soulever des questions légales. Les fonctionnalités de contrôle d'accès d'Iceberg, combinées aux politiques IAM infonuagiques, permettent de restreindre l'accès selon la localisation de l'utilisateur.

Audit de conformité : Les capacités de voyage dans le temps d'Iceberg permettent de démontrer l'historique de localisation des données lors des audits de conformité.

Écosystème de Partenaires Canadiens

Le marché canadien compte plusieurs partenaires et intégrateurs spécialisés dans les déploiements Lakehouse, offrant une expertise locale précieuse.

Les **grands cabinets de conseil** (Deloitte, Accenture, EY) ont développé des pratiques Data & Analytics incluant des compétences Iceberg et Lakehouse. Ces équipes peuvent accompagner les grandes entreprises dans leur transformation, de la stratégie à l'implémentation.

Les **intégrateurs spécialisés** en données offrent une expertise technique approfondie pour les déploiements complexes. Ces firmes comptent souvent des praticiens certifiés sur les principales plateformes Lakehouse.

Les **fournisseurs infonuagiques** avec présence canadienne (AWS, Azure, Google Cloud) offrent tous des services gérés facilitant le déploiement d'Iceberg. Les régions canadiennes garantissent la résidence des données tout en bénéficiant de l'écosystème global de services.

Les **éditeurs de logiciels** comme Dremio, Starburst (Trino) et Confluent proposent des solutions commerciales ajoutant gouvernance, support et fonctionnalités avancées au-dessus des projets open-source.

Défis et Considérations

Complexité Opérationnelle

Malgré ses nombreux avantages, l'adoption d'Apache Iceberg introduit une complexité opérationnelle qui ne doit pas être sous-estimée.

La **maintenance des tables** nécessite une attention continue. Les tables Iceberg accumulent des fichiers de données et des métadonnées au fil des écritures. Sans maintenance régulière, les performances de requête se dégradent et les coûts de stockage augmentent. Les opérations de compaction, d'expiration des snapshots et de réécriture des fichiers de données doivent être planifiées et automatisées.

Les principales opérations de maintenance incluent :

- **Compaction** : Consolidation des petits fichiers de données en fichiers plus grands pour optimiser les performances de lecture. Une table recevant de nombreuses petites écritures peut accumuler des milliers de fichiers minuscules, chacun ajoutant une surcharge lors de la planification des requêtes.
- **Expiration des snapshots** : Suppression des anciens snapshots et des fichiers de métadonnées associés. Sans expiration régulière, l'historique des snapshots croît indéfiniment, augmentant les coûts de stockage et le temps de listage des métadonnées.
- **Réécriture des fichiers de données** : Réorganisation physique des données pour améliorer la localité de partition ou les statistiques. Cette opération, coûteuse en ressources, est nécessaire périodiquement pour maintenir des performances optimales.
- **Nettoyage des fichiers orphelins** : Suppression des fichiers de données et de métadonnées qui ne sont plus référencés par aucun snapshot valide. Ces fichiers peuvent s'accumuler après des échecs d'écriture ou des opérations de maintenance incomplètes.

La **gestion du catalogue** constitue un autre défi. Le catalogue est un composant critique qui doit être hautement disponible et performant. Son échec ou sa dégradation impacte l'ensemble des opérations sur le Lakehouse. Les organisations doivent planifier la haute disponibilité, la sauvegarde et la reprise après sinistre de leur infrastructure de catalogue.

Les considérations clés pour le catalogue incluent :

- **Haute disponibilité** : Le catalogue doit rester accessible même en cas de défaillance partielle de l'infrastructure.
- **Performance** : Les opérations de catalogue (lecture du pointeur de métadonnées, commit des transactions) doivent être rapides, car elles se trouvent sur le chemin critique de chaque requête.
- **Sauvegarde et récupération** : La perte du catalogue peut rendre les tables inaccessibles, même si les données sous-jacentes sont intactes.
- **Cohérence multi-moteur** : Lorsque plusieurs moteurs accèdent aux mêmes tables, le catalogue doit garantir la cohérence des vues.

L'**observabilité** du Lakehouse requiert des métriques et des alertes adaptées. Les indicateurs traditionnels de base de données ne capturent pas toujours les problèmes spécifiques à Iceberg, comme la fragmentation des fichiers, la croissance des métadonnées ou les conflits de concurrence optimiste.

> **Performance : Métriques Clés à Surveiller** Pour maintenir un Lakehouse Iceberg en bonne santé, surveillez activement : - Nombre de fichiers de données par table (alerte si > 10 000 pour les tables non partitionnées) - Taille moyenne des fichiers de données (cible : 128 Mo - 1 Go selon la charge de travail) - Nombre de snapshots actifs (alerte si > 100 sans expiration récente) - Temps de planification des requêtes (détecter la dégradation due à la croissance des métadonnées) - Taux d'échec des commits optimistes (indicateur de contention d'écriture) - Espace consommé par les fichiers orphelins (fuites de stockage)

Courbe d'Apprentissage

L'équipe technique doit acquérir de nouvelles compétences pour exploiter efficacement un Lakehouse Iceberg.

Les **ingénieurs de données** doivent comprendre le modèle de métadonnées d'Iceberg, les implications des différentes stratégies d'écriture (Copy-on-Write vs Merge-on-Read) et les meilleures pratiques de partitionnement. La familiarité avec Spark, Flink ou d'autres moteurs de traitement est souvent un prérequis.

Les **administrateurs de bases de données** habitués aux systèmes traditionnels doivent adapter leurs pratiques. Les concepts de maintenance (compaction, expiration) et de surveillance diffèrent significativement des SGBD classiques.

Les **analystes** et **scientifiques de données** bénéficient généralement d'une transition plus douce, car l'interface SQL reste familière. Cependant, la compréhension des capacités de voyage dans le temps et des performances de partition peut enrichir considérablement leur travail.

Cas Où Iceberg N'est Pas Optimal

Apache Iceberg excelle pour les charges de travail analytiques à grande échelle, mais n'est pas la solution universelle pour tous les problèmes de données.

Les **charges transactionnelles OLTP** avec de nombreuses petites transactions simultanées ne sont pas le cas d'usage cible d'Iceberg. Les systèmes de bases de données relationnelles traditionnels (PostgreSQL,

MySQL) ou les bases de données distribuées (CockroachDB, TiDB) restent plus adaptés pour ces besoins. La surcharge du modèle de métadonnées d'Iceberg ne se justifie pas pour des transactions de quelques millisecondes traitant une poignée de lignes.

Les **données de petite taille** (quelques gigaoctets ou moins) ne bénéficient pas significativement de l'architecture Iceberg. La surcharge des métadonnées et la complexité opérationnelle ne se justifient pas pour des ensembles de données qui tiennent confortablement dans une base de données traditionnelle ou même dans un fichier CSV.

Les **requêtes à très faible latence** (millisecondes) requièrent généralement des systèmes spécialisés comme les bases de données en mémoire ou les caches. Bien que les performances d'Iceberg soient excellentes pour l'analytique, elles ne rivalisent pas avec les systèmes optimisés pour la latence minimale. Pour les tableaux de bord temps réel avec des exigences de latence strictes, des solutions complémentaires comme Apache Druid, Apache Pinot ou ClickHouse peuvent être plus appropriées.

Les **cas d'usage de graphes** avec des traversées complexes de relations sont mieux servis par des bases de données orientées graphes comme Neo4j ou Amazon Neptune. Les tables Iceberg peuvent stocker les données sources, mais les analyses de graphes nécessitent des moteurs spécialisés.

Les **données non structurées** comme les images, les vidéos ou les documents textuels ne sont pas le cas d'usage principal d'Iceberg. Bien qu'Iceberg puisse stocker des références vers ces fichiers, leur traitement requiert généralement des pipelines spécialisés.

La Voie Vers l'Adoption

Pour les organisations qui évaluent l'adoption d'Apache Iceberg, nous recommandons une approche progressive en plusieurs phases :

Phase 1 — Évaluation (1-2 mois) : Identifier les cas d'usage prioritaires, évaluer l'adéquation technique, former une équipe noyau sur les concepts fondamentaux.

Phase 2 — Preuve de concept (2-3 mois) : Implémenter un cas d'usage non critique en environnement de développement, valider les performances et l'intégration avec l'écosystème existant.

Phase 3 — Pilote (3-6 mois) : Déployer en production pour un sous-ensemble de tables, établir les pratiques opérationnelles, développer l'observabilité.

Phase 4 — Expansion (6-18 mois) : Migrer progressivement les charges de travail supplémentaires, optimiser les performances, former les équipes élargies.

Cette approche permet de gérer les risques tout en construisant progressivement les compétences et les pratiques nécessaires au succès à long terme.

Conclusion : Préparer l'Avenir des Données d'Entreprise

L'émergence d'Apache Iceberg et du paradigme Lakehouse représente plus qu'une simple évolution technologique — c'est une redéfinition fondamentale de la façon dont les entreprises gèrent et exploitent leurs actifs de données.

L'Impératif de la Modernisation

En 2025, les organisations qui n'ont pas encore entrepris leur transition vers des architectures ouvertes se trouvent dans une position de désavantage croissant. Les coûts des entrepôts de données propriétaires continuent d'augmenter, tandis que les alternatives ouvertes offrent des fonctionnalités comparables ou supérieures à une fraction du coût. Le verrouillage fournisseur, autrefois un inconvénient accepté, devient un risque stratégique inacceptable dans un environnement technologique qui évolue rapidement.

Les « guerres de formats » des années précédentes touchent à leur fin, avec Apache Iceberg émergeant comme le langage universel des données d'entreprise. Cette convergence simplifie les décisions architecturales : plutôt que de débattre longuement du format optimal, les organisations peuvent concentrer leurs efforts sur l'extraction de valeur de leurs données.

Le Rôle du Lakehouse dans l'Entreprise Agentique

Ce volume s'inscrit dans une monographie plus large sur l'Entreprise Agentique. Le Lakehouse joue un rôle crucial dans cette vision : il fournit la fondation de données fiable et gouvernée sur laquelle les agents d'intelligence artificielle peuvent opérer.

Les agents IA nécessitent des données de haute qualité, cohérentes et accessibles. Le Lakehouse Iceberg fournit précisément cela :

- **Cohérence garantie** : Les transactions ACID assurent que les agents voient toujours un état cohérent des données
- **Gouvernance intégrée** : Le contrôle d'accès et l'audit permettent de tracer les actions des agents
- **Historique complet** : Le voyage dans le temps permet l'analyse de l'évolution des données et la reproductibilité des résultats
- **Interopérabilité** : Les agents peuvent utiliser différents moteurs de calcul selon les besoins de chaque tâche

L'intégration avec Apache Kafka, explorée dans le Volume III et développée dans le chapitre sur le Streaming Lakehouse de ce volume, permet aux agents de réagir aux événements en temps réel tout en ayant accès à l'historique complet des données.

Perspectives Technologiques 2026-2030

Plusieurs tendances émergentes façonneront l'évolution du Lakehouse dans les années à venir.

L'**automatisation intelligente** de la maintenance des tables progressera. Des systèmes de compaction et d'optimisation auto-gérés, guidés par l'apprentissage automatique, réduiront la charge opérationnelle. Les tables s'auto-optimiseront en fonction des patterns d'accès observés.

L'**intégration IA native** deviendra standard. Les Lakehouse fourniront des interfaces directes pour l'entraînement de modèles et l'inférence, éliminant les mouvements de données coûteux vers des systèmes spécialisés.

La **convergence streaming-batch** s'accélérera. La distinction entre traitement en temps réel et traitement par lots deviendra de moins en moins pertinente, le Lakehouse servant de plateforme unifiée pour tous les modes de traitement.

L'**interopérabilité universelle** se renforcera avec des standards comme Apache XTable permettant l'accès transparent aux mêmes données via différents formats de table selon les besoins de chaque moteur.

Recommandations pour les Décideurs

Pour les architectes de données canadiens, les considérations de souveraineté des données et de conformité réglementaire rendent l'adoption de standards ouverts particulièrement pertinente. La capacité de contrôler où les données résident, comment elles sont accédées et qui peut les consulter constitue un avantage concurrentiel dans un contexte réglementaire de plus en plus strict.

Nous recommandons aux organisations de :

1. **Évaluer leur dette technique actuelle** : Inventorier les systèmes de données existants, leurs coûts et leurs limitations
2. **Identifier les cas d'usage prioritaires** : Cibler les projets où le Lakehouse apportera le plus de valeur rapidement
3. **Développer les compétences internes** : Former les équipes sur Iceberg et les technologies associées
4. **Planifier une migration progressive** : Commencer par des tables non critiques pour gagner en expérience
5. **Établir les pratiques opérationnelles** : Définir les procédures de maintenance et de surveillance avant le passage en production

Les chapitres suivants de ce volume exploreront en profondeur l'anatomie technique d'Apache Iceberg, les stratégies de migration depuis les architectures existantes, et les meilleures pratiques opérationnelles pour maintenir un Lakehouse de production. Nous examinerons également l'intégration avec Apache Kafka pour former le Streaming Lakehouse, connectant ainsi ce volume avec les concepts développés dans le Volume III de cette monographie.

L'avenir appartient aux organisations qui embrassent l'ouverture, l'interopérabilité et l'agilité. Apache Iceberg n'est pas seulement un format de table — c'est la fondation sur laquelle les plateformes de données de la prochaine décennie seront construites.

Résumé

Ce chapitre introductif a établi les fondations conceptuelles du Data Lakehouse Apache Iceberg. Les points essentiels à retenir sont organisés autour de six thèmes majeurs.

Évolution Architecturale

Le Lakehouse représente la synthèse des entrepôts de données (gouvernance, transactions ACID, performances SQL) et des lacs de données (stockage économique, formats ouverts, élasticité). Cette convergence n'est pas un compromis mais une véritable avancée architecturale qui élimine les limitations de chaque approche précédente. Apache Iceberg fournit la couche technique qui rend cette synthèse possible, transformant des collections de fichiers Parquet en tables avec des propriétés de base de données.

L'évolution historique — des bases de données relationnelles aux entrepôts dédiés, puis aux lacs de données et enfin au Lakehouse — illustre la quête constante d'un équilibre entre structure et flexibilité, entre performance et économie. Le Lakehouse Iceberg atteint cet équilibre.

Origines et Genèse d'Iceberg

Créé par Netflix en 2017 pour résoudre les limitations d'Apache Hive à l'échelle de l'exaoctet, Iceberg a été conçu avec des objectifs précis : garantir la correction des données par des transactions ACID véritables,

améliorer les performances par des opérations à granularité de fichier, et simplifier l'exploitation des tables analytiques. L'entrepôt de données de Netflix, dépassant les 100 pétaoctets, a servi de banc d'essai pour ces innovations.

Donné à l'Apache Software Foundation en 2018 et devenu projet de premier niveau en 2020, Iceberg bénéficie d'une gouvernance neutre qui a favorisé son adoption massive. Sa conception orientée vers l'interopérabilité, plutôt que vers un écosystème propriétaire, explique pourquoi il s'est imposé comme le standard de facto.

Architecture Technique

Iceberg utilise une hiérarchie de métadonnées à trois niveaux (catalogue, métadonnées, données) qui permet les transactions ACID, l'évolution de schéma sans réécriture, le partitionnement masqué et le voyage dans le temps. Chaque composant joue un rôle précis :

- Le **catalogue** maintient le pointeur vers les métadonnées courantes, garantissant l'atomicité des commits
- Le **fichier de métadonnées** décrit complètement la table : schéma, partitions, snapshots
- Les **listes de manifestes** indexent les manifestes par snapshot, permettant l'élagage au niveau du manifeste
- Les **fichiers manifestes** inventorient les fichiers de données avec leurs statistiques, permettant l'élagage au niveau des fichiers
- Les **fichiers de données** (Parquet, ORC, Avro) contiennent les données réelles

Ce découpage permet des optimisations de requêtes sophistiquées et des commits atomiques même pour des tables de plusieurs pétaoctets.

Écosystème et Adoption

En 2025, Iceberg s'est imposé comme le standard de facto, supporté par tous les principaux fournisseurs infonuagiques (AWS, Google, Azure), plateformes de données (Snowflake, Databricks, Dremio) et moteurs de calcul (Spark, Trino, Flink). Des géants technologiques comme Netflix, Apple, LinkedIn et Airbnb l'utilisent en production pour des charges de travail critiques impliquant des pétaoctets de données.

L'acquisition de Tabular par Databricks et l'open-sourcing de Polaris par Snowflake illustrent la convergence de l'industrie vers Iceberg. Le projet Apache XTable permet l'interopérabilité entre formats, réduisant encore les risques de verrouillage.

Contexte Canadien

Les exigences réglementaires distinctives du Canada (LPRPDE, Loi 25, réglementations sectorielles BSIF) rendent les capacités de gouvernance d'Iceberg particulièrement pertinentes. Le contrôle d'accès granulaire, l'audit via le voyage dans le temps et la flexibilité de résidence des données facilitent la conformité.

Les organisations canadiennes bénéficient de régions infonuagiques locales (Montréal, Toronto, Québec) permettant la souveraineté des données tout en accédant à l'écosystème global de services. L'étude de cas du commerce de détail canadien illustre comment ces capacités se traduisent en avantages concurrentiels tangibles.

Considérations Pratiques

L'adoption d'Iceberg introduit une complexité opérationnelle (maintenance, gestion du catalogue, observabilité) et une courbe d'apprentissage pour les équipes. Les opérations de compaction, d'expiration des snapshots et de nettoyage des fichiers orphelins doivent être planifiées et automatisées.

Iceberg excelle pour l'analytique à grande échelle mais n'est pas optimal pour : - Les charges transactionnelles OLTP avec de nombreuses petites transactions - Les ensembles de données de petite taille (quelques gigaoctets) - Les exigences de latence extrême (millisecondes)

Les chapitres suivants approfondiront l'anatomie technique d'Iceberg, les stratégies de conception architecturale, les approches de migration et les pratiques opérationnelles essentielles pour réussir votre déploiement Lakehouse.

Chapitre suivant : Chapitre IV.2 — Anatomie Technique d'Apache Iceberg

Chapitre IV.2 — Anatomie Technique d'Apache Iceberg

Introduction

La compréhension profonde de l'architecture interne d'Apache Iceberg constitue le fondement de toute implémentation réussie du Data Lakehouse. Alors que le chapitre précédent présentait la valeur stratégique et le positionnement d'Iceberg dans l'écosystème moderne des données, ce chapitre plonge dans les mécanismes techniques qui permettent au format de table de délivrer ses promesses de performance, de fiabilité et d'évolutivité.

Apache Iceberg se distingue des formats de table traditionnels par son architecture métadonnées sophistiquée qui découple complètement la représentation logique des tables de leur disposition physique. Cette séparation fondamentale permet des fonctionnalités qui étaient auparavant impossibles dans les lacs de données : transactions ACID véritables, évolution de schéma sans réécriture, partitionnement masqué et voyage dans le temps. Pour l'architecte de données et l'ingénieur, maîtriser ces mécanismes internes est essentiel pour concevoir des systèmes optimaux et résoudre les problèmes en production.

Dans les architectures de données traditionnelles basées sur Apache Hive, la structure des répertoires servait de métadonnées implicites. Pour déterminer quels fichiers appartiennent à une table, le moteur de requête devait parcourir l'arborescence de fichiers, listant potentiellement des milliers de répertoires de partition et des millions de fichiers de données. Cette approche, bien que simple à comprendre, présente des limitations sévères à grande échelle : temps de planification des requêtes croissant linéairement avec le nombre de partitions, absence de garanties transactionnelles lors des écritures concurrentes, et impossibilité de modifier le schéma de partitionnement sans migration complète des données.

Iceberg résout ces limitations fondamentales en introduisant une couche de métadonnées explicites qui maintient un inventaire complet et versionné de tous les fichiers appartenant à une table. Cette approche de type « manifest » transforme les opérations de planification de requête de complexité O(n) — où n représente le nombre de fichiers ou partitions — en opérations de complexité O(1) en termes d'appels au système de fichiers. Le gain de performance devient spectaculaire à l'échelle : une table de millions de fichiers peut être planifiée aussi rapidement qu'une table de quelques fichiers.

Au-delà de la performance, l'architecture de métadonnées d'Iceberg permet des garanties impossibles avec les approches basées sur le système de fichiers. Les transactions ACID sont implémentées via un protocole de commit optimiste qui garantit l'atomicité des modifications. L'isolation snapshot assure que les lecteurs voient toujours un état cohérent de la table, même pendant les écritures concurrentes. L'historique des snapshots permet le voyage dans le temps et le rollback vers des états antérieurs.

Ce chapitre décortique l'anatomie complète d'une table Iceberg, depuis la couche de catalogue jusqu'aux fichiers de données, en passant par la hiérarchie de métadonnées qui constitue le cœur de l'innovation. Nous explorerons les structures de données détaillées, les algorithmes de commit, les stratégies de mise à jour au niveau des lignes et les mécanismes d'optimisation des requêtes. L'objectif est de fournir aux praticiens une compréhension suffisamment profonde pour prendre des décisions architecturales éclairées et diagnostiquer efficacement les problèmes de performance.

La maîtrise de ces concepts est particulièrement pertinente dans le contexte canadien où les réglementations comme la Loi 25 au Québec et la LPRPDE fédérale imposent des exigences strictes de traçabilité et de gouvernance des données. Les fonctionnalités natives d'Iceberg — voyage dans le temps pour l'audit, suppression au niveau des lignes pour le droit à l'oubli, lignage via les métadonnées — répondent directement à ces exigences réglementaires.

L'Architecture en Trois Couches

Apache Iceberg organise une table selon une architecture en trois couches distinctes, chacune servant un rôle spécifique dans la gestion et l'accès aux données. Cette stratification permet la séparation des préoccupations et offre la flexibilité nécessaire pour supporter différents moteurs de calcul et systèmes de stockage.

Vue d'Ensemble de l'Architecture

La première couche, le catalogue, agit comme point d'entrée et source de vérité pour localiser les tables. La deuxième couche, les métadonnées, contient toute l'information nécessaire pour comprendre l'état d'une table à n'importe quel moment. La troisième couche, les données, stocke les enregistrements réels dans des formats de fichiers optimisés.

Cette architecture se différencie fondamentalement de l'approche traditionnelle de type Hive où la structure des répertoires servait de métadonnées implicites. Dans le modèle Hive, pour déterminer quels fichiers appartiennent à une table, le moteur de requête devait effectuer des opérations coûteuses de listage de répertoires, parcourant potentiellement des milliers de partitions et des millions de fichiers. Iceberg remplace cette approche par un système de manifestes explicites qui énumère précisément chaque fichier appartenant à la table.

Flux de Lecture d'une Table

Lorsqu'un moteur de requête accède à une table Iceberg, le flux d'opérations suit une séquence bien définie. Premièrement, le moteur interroge le catalogue pour obtenir l'emplacement du fichier de métadonnées courant. Deuxièmement, il lit ce fichier metadata.json pour identifier le snapshot actif. Troisièmement, il accède à la liste de manifestes de ce snapshot. Quatrièmement, il consulte les fichiers manifestes pour déterminer quels fichiers de données doivent être lus. Finalement, il lit uniquement les fichiers de données nécessaires.

Cette approche transforme une opération de complexité O(n) — où n représente le nombre de fichiers ou partitions — en une opération de complexité O(1) en termes d'appels distants pour la planification des requêtes. Le gain de performance devient spectaculaire à grande échelle : une table de millions de fichiers peut être planifiée aussi rapidement qu'une table de quelques fichiers.

La Couche de Catalogue

Le catalogue Iceberg remplit deux fonctions essentielles : le suivi des tables par nom et la gestion atomique des pointeurs vers les métadonnées courantes. Sans catalogue, chaque moteur de requête devrait connaître l'emplacement exact des fichiers de métadonnées, rendant impossible la coordination entre différents systèmes.

Rôle et Responsabilités

Le catalogue maintient une correspondance entre les noms de tables et les emplacements des fichiers de métadonnées. Lorsqu'une table est mise à jour, le catalogue doit atomiquement basculer le pointeur vers le nouveau fichier de métadonnées. Cette atomicité est cruciale : elle garantit que tous les lecteurs voient soit l'ancien état complet, soit le nouveau état complet, jamais un état intermédiaire incohérent.

Les responsabilités du catalogue incluent la création et la suppression de tables, le renommage de tables entre espaces de noms, la gestion des espaces de noms eux-mêmes, et le support des opérations de commit avec détection de conflits. Le catalogue ne stocke pas les métadonnées de la table elle-même — celles-ci résident dans les fichiers metadata.json — mais uniquement le pointeur vers ces métadonnées.

Types de Catalogues

Iceberg supporte plusieurs implémentations de catalogues, classées en deux catégories : les catalogues basés sur fichiers et les catalogues basés sur services.

Les catalogues basés sur fichiers utilisent un fichier version-hint.txt pour pointer vers le fichier metadata.json courant. Le catalogue Hadoop en est l'exemple canonique. Cette approche est simple mais présente des limitations pour les environnements à haute concurrence, car les systèmes de fichiers distribués ne garantissent pas tous l'atomicité des opérations de renommage nécessaires aux commits.

Les catalogues basés sur services maintiennent les références dans une base de données ou un service externe. Le Hive Metastore, AWS Glue Data Catalog, et les implémentations REST en sont des exemples. Ces catalogues offrent de meilleures garanties de concurrence et des fonctionnalités avancées comme le contrôle d'accès et l'audit.

La Spécification REST Catalog

La communauté Iceberg a développé une spécification REST Catalog pour standardiser les interactions avec n'importe quel catalogue. Cette spécification OpenAPI définit les points de terminaison requis pour les opérations sur les tables et les espaces de noms.

Les avantages du REST Catalog sont considérables. Un client unique peut interagir avec n'importe quel catalogue implémentant la spécification, éliminant le besoin d'implémentations spécifiques pour chaque langage et chaque catalogue. Cette interopérabilité simplifie l'adoption multi-moteurs et facilite les migrations entre fournisseurs.

Les points de terminaison principaux de la spécification incluent GET /v1/config pour la configuration du catalogue, POST /v1/namespaces pour la création d'espaces de noms, POST /v1/namespaces/{namespace}/tables pour la création de tables, et POST /v1/namespaces/{namespace}/tables/{table} pour les commits de métadonnées.

Des implémentations de catalogues REST comme Apache Polaris (en incubation), Nessie, et les services gérés d'AWS Glue et Snowflake adoptent cette spécification. La convergence vers ce standard renforce l'interopérabilité de l'écosystème Iceberg.

> **Performance**
> L'utilisation d'un catalogue REST avec mise en cache côté serveur peut réduire la latence de planification des requêtes de 40 à 60 % par rapport aux catalogues basés sur fichiers, particulièrement pour les tables à haute fréquence d'accès.

La Couche de Métadonnées

La couche de métadonnées constitue le cœur de l'innovation d'Iceberg. Elle transforme une collection de fichiers en une table cohérente avec des propriétés ACID, un historique de versions et des statistiques pour l'optimisation des requêtes. Cette couche comprend quatre types de fichiers : le fichier de métadonnées principal (metadata.json), les listes de manifestes, les fichiers manifestes et les fichiers Puffin pour les statistiques avancées.

Le Fichier metadata.json

Le fichier metadata.json est le point d'entrée pour comprendre l'état d'une table. Il contient toute l'information nécessaire pour reconstruire la table à un moment donné, incluant l'historique des schémas, les spécifications de partitionnement, l'historique des snapshots et les propriétés de la table.

La structure d'un fichier metadata.json typique comprend les champs suivants :

```
{
  "format-version": 2,
  "table-uuid": "f7m3-a812-4a5c-96b6-8a3a",
  "location": "s3://warehouse/db/events",
  "last-updated-ms": 1664472000000,
  "last-column-id": 5,
  "schemas": [...],
  "current-schema-id": 0,
  "partition-specs": [...],
  "default-spec-id": 0,
  "sort-orders": [...],
  "default-sort-order-id": 0,
  "snapshots": [...],
  "current-snapshot-id": 8235603094578364387,
  "snapshot-log": [...],
  "metadata-log": [...]
}
```

Le champ format-version indique la version de la spécification Iceberg. La version 1 supporte les tables analytiques de base, la version 2 ajoute les opérations au niveau des lignes avec les fichiers de suppression, et la version 3 (adoptée en 2024-2025) introduit les vecteurs de suppression, les nouveaux types de données comme variant et les types géospatiaux.

Le champ table-uuid identifie uniquement la table à travers les opérations. Si un client détecte un UUID différent après rafraîchissement, cela indique une corruption ou un conflit nécessitant une investigation.

Le champ location spécifie l'emplacement de base où sont stockés les fichiers de données, les manifestes et les métadonnées. Les écrivains utilisent cet emplacement pour déterminer où placer les nouveaux fichiers.

L'historique des schémas dans le tableau schemas préserve chaque version de schéma avec un identifiant unique. Cette préservation permet aux lecteurs d'interpréter correctement les fichiers de données écrits avec d'anciens schémas.

Les Snapshots

Un snapshot représente l'état complet d'une table à un moment précis. Chaque opération d'écriture — insertion, mise à jour, suppression — crée un nouveau snapshot. Cette immuabilité des snapshots est fondamentale pour les garanties ACID et le voyage dans le temps.

La structure d'un snapshot comprend :

```
{
  "snapshot-id": 8235603094578364387,
  "parent-snapshot-id": 3051729675574644887,
  "timestamp-ms": 1664472000000,
  "summary": {
    "operation": "append",
    "added-data-files": "10",
    "added-records": "50000",
    "total-data-files": "150",
    "total-records": "750000"
  },
  "manifest-list": "s3://warehouse/db/events/metadata/snap-8235603.avro",
  "schema-id": 0
}
```

Le champ parent-snapshot-id établit la lignée des snapshots, permettant de reconstituer l'historique complet des modifications. Le champ summary fournit des statistiques sur l'opération effectuée, utiles pour le monitoring et le débogage.

Chaque snapshot référence une liste de manifestes qui décrit l'ensemble exact des fichiers appartenant à la table à ce moment. Cette référence est le mécanisme qui permet l'isolation des lectures : un lecteur utilise le snapshot qui était courant au moment où il a chargé les métadonnées et n'est pas affecté par les écritures concurrentes.

Les Listes de Manifestes

La liste de manifestes (manifest list) est un fichier Avro qui énumère tous les fichiers manifestes composant un snapshot. Elle contient également des statistiques agrégées sur chaque manifeste, permettant un premier niveau d'élagage lors de la planification des requêtes.

Chaque entrée de la liste de manifestes inclut :

```
{
  "manifest_path": "s3://warehouse/db/events/metadata/d2f5ebe2.avro",
  "manifest_length": 7292,
  "partition_spec_id": 0,
  "added_snapshot_id": 8235603094578364387,
  "added_data_files_count": 10,
  "existing_data_files_count": 140,
  "deleted_data_files_count": 0,
  "added_rows_count": 50000,
  "existing_rows_count": 700000,
  "deleted_rows_count": 0,
  "partitions": [
    {
      "contains_null": false,
      "contains_nan": false,
      "lower_bound": "2024-01-01",
      "upper_bound": "2024-01-31"
    }
  ]
}
```

Les statistiques de partition (partitions) permettent d'éliminer des manifestes entiers sans les lire. Si une requête filtre sur une date hors de la plage d'un manifeste, ce manifeste peut être ignoré complètement. Ce premier niveau d'élagage réduit considérablement le volume de métadonnées à lire pour les requêtes sélectives.

Les Fichiers Manifestes

Les fichiers manifestes sont le niveau le plus granulaire de la hiérarchie de métadonnées. Chaque manifeste est un fichier Avro qui liste un sous-ensemble des fichiers de données de la table, avec des statistiques détaillées pour chaque fichier.

La structure d'une entrée de manifeste comprend :

```
{
  "status": 1,
  "snapshot_id": 8235603094578364387,
  "data_file": {
    "file_path": "s3://warehouse/db/events/data/part-00001.parquet",
    "file_format": "PARQUET",
    "partition": {"event_date": "2024-01-15"},
    "record_count": 5000,
    "file_size_in_bytes": 15728640,
    "column_sizes": {1: 1048576, 2: 524288, 3: 2097152},
    "value_counts": {1: 5000, 2: 5000, 3: 4850},
    "null_value_counts": {1: 0, 2: 0, 3: 150},
    "nan_value_counts": {},
    "lower_bounds": {1: "event_001", 2: 100, 3: 1.5},
    "upper_bounds": {1: "event_999", 2: 9999, 3: 99.9}
  }
}
```

Le champ status indique l'état du fichier dans ce manifeste : 0 pour existant, 1 pour ajouté, 2 pour supprimé. Cette information est cruciale pour le suivi des modifications entre snapshots.

Les statistiques au niveau des colonnes — lower_bounds et upper_bounds — permettent l'élagage de fichiers (data skipping). Si une requête recherche des événements avec un identifiant supérieur à « event_999 », ce fichier peut être ignoré car sa borne supérieure est exactement « event_999 ».

Les comptages de valeurs null et NaN aident également l'optimiseur. Une colonne avec 100 % de valeurs null n'a pas besoin d'être lue si la requête filtre sur des valeurs non-null.

> **Performance**
> Les statistiques de manifeste permettent typiquement d'éliminer 70 à 95 % des fichiers pour les requêtes sélectives, transformant des scans de téraoctets en lectures de gigaoctets.

La Couche de Données

La couche de données stocke les enregistrements réels dans des formats de fichiers optimisés pour l'analytique. Iceberg est agnostique au format de fichier et supporte Apache Parquet, Apache ORC et Apache Avro.

Formats de Fichiers Supportés

Apache Parquet est le format par défaut et le plus largement utilisé. Son organisation en colonnes, sa compression efficace et ses statistiques de pied de page en font un choix optimal pour les charges analytiques. Parquet supporte des schémas complexes avec des structures imbriquées, des listes et des maps.

La structure interne de Parquet est particulièrement bien adaptée à Iceberg. Chaque fichier Parquet est organisé en groupes de lignes (row groups), typiquement de 128 Mo, qui peuvent être lus indépendamment. Chaque groupe de lignes contient des métadonnées incluant les statistiques min/max par colonne, permettant un élagage supplémentaire au-delà des statistiques de manifeste Iceberg.

Les colonnes dans Parquet sont stockées consécutivement, permettant de ne lire que les colonnes nécessaires à une requête (projection pushdown). La compression est appliquée par colonne avec des codecs adaptés aux types de données : dictionnaire pour les chaînes répétitives, delta encoding pour les entiers séquentiels, run-length encoding pour les valeurs répétées.

Apache ORC offre des caractéristiques similaires avec une compression souvent supérieure et des index intégrés plus sophistiqués. ORC maintient des index de bloom filter et des index de position qui peuvent accélérer les recherches ponctuelles. Il est particulièrement populaire dans l'écosystème Hive et pour les charges avec des patterns d'accès prédictibles.

Apache Avro, bien que principalement orienté lignes, est utilisé pour les fichiers de métadonnées (manifestes et listes de manifestes) et peut servir pour les données dans certains cas d'usage nécessitant une évolution de schéma flexible ou une sérialisation compacte pour le streaming.

Configuration des Formats de Fichiers

Les propriétés de table permettent de configurer finement le comportement d'écriture :

```
-- Configuration Parquet avancée
ALTER TABLE events SET TBLPROPERTIES (
  'write.format.default' = 'parquet',
  'write.parquet.compression-codec' = 'zstd',
  'write.parquet.compression-level' = '3',
  'write.parquet.dict-size-bytes' = '2097152',
  'write.parquet.page-size-bytes' = '1048576',
  'write.parquet.row-group-size-bytes' = '134217728',
  'write.parquet.bloom-filter-enabled.column.user_id' = 'true'
);

-- Configuration ORC
ALTER TABLE events SET TBLPROPERTIES (
  'write.format.default' = 'orc',
  'write.orc.compression-codec' = 'zstd',
  'write.orc.stripe-size-bytes' = '67108864',
  'write.orc.bloom.filter.columns' = 'user_id,product_id'
);
```

Immutabilité des Fichiers

Un principe fondamental d'Iceberg est l'immutabilité des fichiers. Une fois écrit, un fichier de données n'est jamais modifié. Les mises à jour et suppressions sont gérées soit par réécriture complète du fichier (Copy-on-Write), soit par des fichiers de suppression séparés (Merge-on-Read).

Cette immutabilité simplifie considérablement la gestion de la concurrence et la récupération après erreur. Il n'y a jamais de fichiers partiellement écrits ou corrompus par des modifications concurrentes. Les lecteurs peuvent accéder aux fichiers en toute sécurité sans verrous.

L'immutabilité facilite également le caching. Un fichier une fois lu peut être mis en cache indéfiniment car son contenu ne changera jamais. Les systèmes de stockage peuvent optimiser la réplication et la distribution en sachant que les fichiers sont stables.

Organisation Physique

Contrairement aux tables Hive traditionnelles, Iceberg ne dépend pas de la structure des répertoires pour organiser les données. Les fichiers d'une même partition peuvent résider dans différents répertoires ou préfixes. Cette flexibilité permet d'éviter les goulots d'étranglement liés à la limitation de débit des systèmes de stockage objet qui peuvent throttler les requêtes sur un préfixe unique.

En pratique, de nombreuses implémentations organisent tout de même les fichiers par partition pour faciliter la navigation manuelle et le débogage, mais cette organisation n'est pas requise par la spécification. Certaines architectures utilisent intentionnellement une distribution aléatoire des fichiers pour maximiser le parallélisme de lecture sur les systèmes de stockage objet.

Tri des Données

Iceberg supporte le tri des données au sein des fichiers pour améliorer l'efficacité des requêtes. Un ordre de tri (sort order) peut être défini sur la table :

```
-- Définition d'un ordre de tri
ALTER TABLE events WRITE ORDERED BY event_date DESC, event_type ASC;

-- Réécriture avec tri pour optimiser les lectures
CALL catalog.system.rewrite_data_files(
  table => 'db.events',
  strategy => 'sort',
  sort_order => 'event_date DESC, event_type ASC'
);
```

Les données triées améliorent l'efficacité de la compression (valeurs similaires groupées) et permettent un élagage plus agressif via les statistiques min/max. Pour les requêtes qui filtrent sur les colonnes de tri, le bénéfice peut être substantiel.

Gestion des Schémas

L'évolution de schéma est l'une des fonctionnalités les plus puissantes d'Iceberg. Elle permet de modifier la structure d'une table — ajouter, supprimer, renommer ou réordonner des colonnes — sans réécrire les données existantes.

Le Système d'Identifiants de Colonnes

Le mécanisme clé qui permet l'évolution de schéma sans effets de bord est l'utilisation d'identifiants uniques pour chaque colonne. Contrairement aux systèmes qui identifient les colonnes par leur position ou leur nom, Iceberg assigne un entier unique permanent à chaque champ lors de sa création.

Considérons un fichier écrit avec le schéma suivant : 1: a int, 2: b string, 3: c double. Si le schéma évolue pour devenir 3: measurement double, 2: name string, 4: d boolean, la projection de lecture reste correcte. La colonne c (ID 3) est maintenant appelée measurement, la colonne b (ID 2) est renommée name, et la nouvelle colonne d (ID 4) retourne null pour les anciens fichiers.

Ce système garantit que : - L'ajout d'une colonne ne lit jamais les valeurs d'une autre colonne existante - La suppression d'une colonne ne modifie pas les valeurs des autres colonnes - Le renommage d'une colonne n'affecte pas les données sous-jacentes - La réorganisation des colonnes ne change pas les valeurs associées

Opérations d'Évolution de Schéma

Iceberg supporte les opérations suivantes sur les schémas :

Add — Ajoute une nouvelle colonne à la table ou à une structure imbriquée. La colonne reçoit un nouvel identifiant unique. Les fichiers existants retournent null ou la valeur par défaut pour cette colonne.

Drop — Supprime une colonne existante. Les fichiers existants conservent les données de cette colonne mais elles ne sont plus accessibles. La suppression est une opération de métadonnées uniquement.

Rename — Renomme une colonne ou un champ imbriqué. L'identifiant reste le même, seul le nom change dans le schéma.

Reorder — Modifie l'ordre des colonnes. Comme les colonnes sont identifiées par leur ID, l'ordre n'affecte pas la lecture des fichiers existants.

Update — Élargit le type d'une colonne. Les promotions de type supportées incluent : - int vers long - float vers double - decimal vers decimal avec précision plus large

```
-- Exemples d'évolution de schéma en SQL
ALTER TABLE events ADD COLUMN user_agent STRING;
ALTER TABLE events DROP COLUMN legacy_field;
ALTER TABLE events RENAME COLUMN old_name TO new_name;
ALTER TABLE events ALTER COLUMN amount TYPE decimal(12,2);
```

Valeurs par Défaut

La spécification Iceberg v3 introduit un support amélioré pour les valeurs par défaut. Deux types de défauts existent : initial-default et write-default.

Le initial-default est utilisé lors de la lecture de fichiers écrits avant l'ajout de la colonne. Une fois défini, il ne peut pas être modifié.

Le write-default est utilisé lors de l'écriture si aucune valeur n'est fournie. Il peut être modifié au fil du temps.

Cette distinction permet des scénarios où une colonne a une valeur par défaut différente pour les données historiques et les nouvelles données. Par exemple, une colonne status pourrait avoir une valeur par défaut "UNKNOWN" pour les données historiques (initial-default) mais "PENDING" pour les nouvelles insertions (write-default).

Promotions de Type Sûres

Iceberg supporte l'élargissement sécurisé de certains types sans réécriture de données. Les promotions autorisées sont conçues pour être toujours sans perte :

Type Source	Type Cible	Notes
int	long	Élargissement entier standard
float	double	Élargissement flottant standard
decimal(P1, S)	decimal(P2, S)	P2 > P1, même échelle

Les promotions non supportées incluent les conversions string vers numeric, les réductions de précision, et les changements de type incompatibles. Ces restrictions garantissent qu'aucune donnée existante ne sera mal interprétée après l'évolution du schéma.

```
-- Promotion de type sécurisée
ALTER TABLE events ALTER COLUMN quantity TYPE BIGINT;  -- int -> long

-- Promotion de decimal
ALTER TABLE events ALTER COLUMN amount TYPE DECIMAL(12, 2);  -- de DECIMAL(10,2)
```

Schémas Imbriqués

L'évolution de schéma dans Iceberg s'applique également aux structures imbriquées. Les structs, arrays et maps peuvent voir leurs champs internes modifiés avec les mêmes garanties de sécurité que les colonnes de premier niveau.

```
-- Ajout d'un champ à un struct imbriqué
ALTER TABLE events ALTER COLUMN address ADD COLUMN country STRING;

-- Renommage d'un champ de struct
ALTER TABLE events ALTER COLUMN address.city RENAME TO city_name;
```

Chaque champ dans une structure imbriquée possède son propre identifiant unique, permettant une évolution indépendante à n'importe quel niveau de la hiérarchie.

Gestion du Partitionnement

Le partitionnement dans Iceberg représente une innovation majeure par rapport aux systèmes traditionnels. Le concept de partitionnement masqué (hidden partitioning) libère les utilisateurs de la nécessité de comprendre et de gérer explicitement la disposition physique des données.

Partitionnement Masqué

Dans les systèmes traditionnels comme Hive, le partitionnement est explicite. Les partitions apparaissent comme des colonnes dans le schéma et doivent être spécifiées lors des écritures et des requêtes. Cette approche génère plusieurs problèmes : les producteurs de données doivent calculer correctement les valeurs de partition, les consommateurs doivent connaître le schéma de partitionnement pour écrire des requêtes efficaces, et les erreurs de partitionnement corrompent silencieusement les données.

Iceberg résout ces problèmes en gérant le partitionnement comme une transformation de métadonnées. Les valeurs de partition sont dérivées automatiquement des colonnes source selon des transformations configurées. Les producteurs écrivent simplement les données, Iceberg calcule les partitions. Les consommateurs écrivent des requêtes naturelles, Iceberg applique automatiquement l'élagage de partition.

Transformations de Partition

Iceberg supporte plusieurs transformations pour dériver les valeurs de partition :

Identity — Utilise la valeur de la colonne directement. Approprié pour les colonnes catégoriques à faible cardinalité.

Year, Month, Day, Hour — Extrait les composantes temporelles d'une colonne timestamp ou date. Permet de partitionner par granularité temporelle sans colonnes dérivées explicites.

Bucket(N) — Distribue les valeurs en N seaux par hachage. Utile pour les colonnes à haute cardinalité comme les identifiants, assurant une distribution uniforme.

Truncate(W) — Tronque les valeurs à une largeur fixe. Pour les chaînes, conserve les W premiers caractères. Pour les nombres, arrondit aux bins de taille W.

```
-- Création d'une table avec partitionnement masqué
CREATE TABLE events (
  event_id STRING,
  event_time TIMESTAMP,
  user_id BIGINT,
  event_type STRING
)
USING iceberg
PARTITIONED BY (
  day(event_time),
  bucket(16, user_id)
);

-- Requête naturelle - l'élagage s'applique automatiquement
SELECT * FROM events
WHERE event_time BETWEEN '2024-01-01' AND '2024-01-15'
  AND user_id = 12345;
```

Dans cet exemple, le moteur de requête comprend que le filtre sur event_time implique les partitions de jours du 1er au 15 janvier, et que user_id = 12345 correspond à un seau spécifique. Les fichiers des autres partitions sont ignorés sans que l'utilisateur n'ait à spécifier ces filtres explicitement.

Évolution du Partitionnement

L'évolution du partitionnement permet de modifier le schéma de partitionnement d'une table existante sans réécrire les données. Lorsque la spécification de partition change, les anciennes données conservent leur partitionnement original tandis que les nouvelles données utilisent le nouveau schéma.

Chaque spécification de partition reçoit un identifiant unique, et les fichiers manifestes enregistrent quelle spécification était active lors de l'écriture de chaque fichier. Lors de la planification des requêtes, Iceberg dérive les filtres appropriés pour chaque layout de partition.

```
-- Migration du partitionnement journalier vers horaire
ALTER TABLE events ADD PARTITION FIELD hour(event_time);

-- Les deux schémas coexistent
-- Données 2023: partitionnées par jour
-- Données 2024: partitionnées par jour ET par heure
```

Cette flexibilité est cruciale pour les tables à longue durée de vie dont les patterns d'accès évoluent. Une table initialement partitionnée par mois peut être migrée vers un partitionnement journalier quand le volume augmente, sans migration de données coûteuse.

La planification des requêtes avec plusieurs spécifications de partition fonctionne comme suit. Supposons une table initialement partitionnée par mois(event_time), puis modifiée pour être partitionnée par jour(event_time). Une requête filtrant sur event_time BETWEEN "2024-01-15" AND "2024-01-20" génère deux ensembles de filtres : pour les fichiers anciens avec la partition mois, le filtre devient event_month = "2024-01"; pour les fichiers nouveaux avec la partition jour, le filtre devient event_day IN ("2024-01-15", "2024-01-16", ..., "2024-01-20"). Chaque ensemble est appliqué aux manifestes correspondants.

Bonnes Pratiques de Partitionnement

Le choix du schéma de partitionnement impacte significativement les performances. Plusieurs principes guident ce choix.

Analyser les patterns de requête — Le partitionnement doit correspondre aux filtres les plus fréquents. Si 90 % des requêtes filtrent par date, le partitionnement temporel est approprié. Si les requêtes filtrent principalement par région, le partitionnement par région est préférable.

Équilibrer la granularité — Trop peu de partitions (par exemple, par année pour des données quotidiennes) ne permet pas un élagage efficace. Trop de partitions (par exemple, par seconde) crée une explosion de métadonnées. Une bonne règle est de viser des partitions contenant entre 100 Mo et 1 Go de données.

Éviter les colonnes à haute cardinalité en identity — Partitionner par identity sur une colonne avec des millions de valeurs uniques crée autant de partitions. Utiliser plutôt bucket() ou truncate() pour regrouper les valeurs.

Considérer les jointures — Si deux tables sont fréquemment jointes, un partitionnement compatible peut permettre des jointures partition-à-partition plus efficaces.

> **Migration**
> *De* : Partitionnement Hive rigide avec colonnes explicites
> *Vers* : Partitionnement masqué Iceberg avec transformations
> *Stratégie* : Utiliser les métadonnées de migration Iceberg pour convertir les tables Hive existantes. Les partitions Hive sont préservées et Iceberg applique automatiquement l'élagage basé sur les filtres de requête.

Opérations au Niveau des Lignes

La version 2 de la spécification Iceberg introduit le support des opérations au niveau des lignes — UPDATE et DELETE — pour les tables analytiques composées de fichiers immuables. Deux stratégies sont disponibles : Copy-on-Write (COW) et Merge-on-Read (MOR), avec différents compromis entre performance d'écriture et de lecture.

Copy-on-Write (COW)

Dans la stratégie Copy-on-Write, toute modification d'une ligne provoque la réécriture complète du fichier de données contenant cette ligne. Si un fichier de 1 Go contient une ligne à mettre à jour, un nouveau fichier de 1 Go est créé avec toutes les lignes sauf celle modifiée, plus la ligne mise à jour.

Avantages de COW : - Performance de lecture optimale — aucune réconciliation nécessaire lors des requêtes - Pas de fichiers de suppression à gérer - Modèle mental simple

Inconvénients de COW : - Amplification d'écriture significative pour les modifications ponctuelles - Coût élevé pour les tables à haute fréquence de mise à jour - Latence d'écriture proportionnelle à la taille des fichiers modifiés

COW est recommandé pour les charges avec lectures fréquentes et mises à jour peu fréquentes, ou pour les mises à jour par lots massifs où la réécriture est inévitable.

Merge-on-Read (MOR)

La stratégie Merge-on-Read évite la réécriture immédiate en créant des fichiers de suppression qui enregistrent les lignes invalidées. Lors de la lecture, ces fichiers de suppression sont appliqués pour filtrer les lignes obsolètes des fichiers de données originaux.

Iceberg supporte deux types de fichiers de suppression :

Position Deletes — Enregistrent le chemin du fichier de données et la position ordinale de la ligne supprimée. La structure est simple : (file_path, row_position). Pour supprimer la ligne 42 du fichier part-00001.parquet, le fichier de position delete contient (« part-00001.parquet », 42).

```
+--------------------------------+----------+
| file_path                      | pos      |
+--------------------------------+----------+
| .../part-00001.parquet         | 42       |
| .../part-00001.parquet         | 157      |
| .../part-00003.parquet         | 891      |
+--------------------------------+----------+
```

Equality Deletes — Identifient les lignes à supprimer par les valeurs de certaines colonnes plutôt que par position. Par exemple, « supprimer où order_id = 12345 ». Cette approche est plus rapide à écrire car elle ne nécessite pas de lire les fichiers existants pour déterminer les positions.

```
+-------------------+
| order_id          |
+-------------------+
| 12345             |
| 67890             |
+-------------------+
```

Les equality deletes utilisent des numéros de séquence pour garantir que seules les lignes écrites avant la suppression sont affectées. Une ligne avec order_id = 12345 écrite après le delete equality n'est pas supprimée.

Avantages de MOR : - Écritures rapides — seuls les changements sont écrits - Faible latence pour les mises à jour ponctuelles - Adapté aux charges à haute fréquence de modification

Inconvénients de MOR : - Overhead de lecture pour appliquer les suppressions - Nécessite une compaction régulière pour maintenir les performances - Complexité accrue de la gestion des fichiers

Vecteurs de Suppression (Version 3)

La version 3 de la spécification introduit les vecteurs de suppression (deletion vectors), une représentation plus efficace des suppressions par position. Au lieu de fichiers séparés, les suppressions sont encodées dans des bitmaps binaires stockés dans des fichiers Puffin.

Les vecteurs de suppression offrent plusieurs avantages sur les fichiers de position delete : - Un seul vecteur par fichier de données maximum, simplifiant la gestion - Représentation bitmap compacte pour les suppressions denses - Lecture plus efficace car le vecteur peut être mis en cache

La migration vers les vecteurs de suppression est recommandée pour les tables v3 avec des charges de mise à jour significatives.

Configuration des Stratégies

Iceberg permet de configurer indépendamment la stratégie pour les opérations DELETE, UPDATE et MERGE :

```
-- Configuration de table pour MOR
ALTER TABLE events SET TBLPROPERTIES (
  'write.delete.mode' = 'merge-on-read',
  'write.update.mode' = 'merge-on-read',
  'write.merge.mode' = 'copy-on-write'
);
```

Cette flexibilité permet d'optimiser selon les patterns d'utilisation. Par exemple, utiliser MOR pour les suppressions fréquentes de conformité RGPD et COW pour les mises à jour par lots massifs.

> **Étude de cas : Institution Financière Canadienne**
> *Secteur* : Services financiers
> *Défi* : Conformité aux demandes de suppression LPRPDE avec tables de 50 To
> *Solution* : Configuration MOR pour les suppressions, compaction nocturne
> *Résultats* : Temps de suppression réduit de 4 heures (COW) à 5 minutes (MOR), avec impact négligeable sur les performances de lecture après compaction quotidienne

Statistiques et Optimisation des Requêtes

Apache Iceberg collecte et maintient plusieurs niveaux de statistiques qui permettent aux moteurs de requête d'optimiser l'exécution. Ces statistiques vont des métriques simples dans les manifestes aux structures probabilistes sophistiquées dans les fichiers Puffin.

Statistiques de Manifeste

Chaque entrée de manifeste contient des statistiques au niveau des colonnes pour le fichier de données référencé :

Comptages — record_count indique le nombre total d'enregistrements, value_counts le nombre de valeurs non-null par colonne, null_value_counts le nombre de null par colonne, et nan_value_counts le nombre de NaN pour les colonnes flottantes.

Bornes — lower_bounds et upper_bounds stockent les valeurs minimales et maximales pour chaque colonne. Ces bornes sont sérialisées en binaire selon le type de données.

Tailles — column_sizes indique la taille en octets de chaque colonne dans le fichier, et file_size_in_bytes la taille totale du fichier.

Ces statistiques permettent l'élagage de fichiers (data skipping). Pour une requête avec WHERE amount > 1000, les fichiers dont la borne supérieure de la colonne amount est inférieure ou égale à 1000 peuvent être ignorés sans être lus.

Filtres de Bloom

Les filtres de Bloom sont des structures probabilistes qui permettent de tester rapidement si une valeur appartient à un ensemble. Ils sont particulièrement utiles pour les colonnes à haute cardinalité où les bornes min/max sont peu sélectives.

Pour activer les filtres de Bloom lors de l'écriture :

```
-- Activation des filtres de Bloom pour une colonne
ALTER TABLE events SET TBLPROPERTIES (
  'write.parquet.bloom-filter-enabled.column.user_id' = 'true',
  'write.parquet.bloom-filter-max-bytes' = '1048576'
);
```

Un filtre de Bloom peut répondre à la question « cette valeur est-elle possiblement dans ce fichier ? » avec deux réponses possibles : « certainement non » ou « peut-être oui ». Cette propriété permet d'éliminer des fichiers avec certitude sans faux négatifs, au prix de quelques faux positifs qui nécessiteront une lecture du fichier.

Fichiers Puffin et Statistiques Avancées

Le format Puffin est conçu pour stocker des statistiques et index additionnels qui ne peuvent pas être directement intégrés dans les manifestes. Son cas d'usage principal est le stockage des estimations du nombre de valeurs distinctes (NDV) utilisant l'algorithme Theta Sketch.

La structure d'un fichier Puffin comprend : - Un en-tête magique identifiant le format - Des blobs contenant les statistiques sérialisées - Un pied de page avec les métadonnées des blobs

Le blob type apache-datasketches-theta-v1 stocke un sketch Theta qui permet d'estimer le NDV d'une colonne avec une précision configurable. Cette information est cruciale pour l'optimisation des jointures : connaître qu'une colonne a 10 valeurs distinctes versus 10 millions permet au planificateur de choisir la bonne stratégie de jointure.

```
# Génération des statistiques Puffin avec Spark
spark.sql("CALL catalog.system.compute_table_stats('events')")
```

Les statistiques Puffin sont liées à un snapshot spécifique et doivent être régénérées après des modifications significatives de la table. AWS Glue Data Catalog et d'autres services offrent une génération automatisée de ces statistiques.

Élagage Multi-Niveau

Lors de la planification d'une requête, Iceberg applique l'élagage à plusieurs niveaux :

1. **Élagage de partition** — Utilise les statistiques de la liste de manifestes pour éliminer des manifestes entiers dont les plages de partition ne correspondent pas aux filtres de requête.
2. **Élagage de fichier** — Pour les manifestes restants, utilise les bornes de colonnes pour éliminer les fichiers dont les valeurs sont hors de la plage recherchée.
3. **Élagage de groupe de lignes** — Au sein des fichiers Parquet sélectionnés, utilise les statistiques de pied de page pour éliminer des groupes de lignes (row groups).
4. **Élagage par Bloom** — Pour les colonnes avec filtres de Bloom activés, vérifie si les valeurs recherchées sont possiblement présentes.

Cette cascade d'élagage peut réduire le volume de données lu de plusieurs ordres de grandeur pour les requêtes sélectives.

Transactions et Concurrence

Iceberg fournit des garanties ACID complètes pour les opérations sur les tables, incluant l'atomicité, la cohérence, l'isolation et la durabilité. Ces garanties sont essentielles pour les environnements de production où plusieurs processus accèdent simultanément aux mêmes tables.

Isolation des Lectures

Les lectures dans Iceberg sont toujours isolées des écritures concurrentes. Lorsqu'un lecteur charge les métadonnées d'une table, il obtient une référence au snapshot courant. Toutes les opérations de ce lecteur utilisent ce snapshot, même si de nouveaux snapshots sont créés pendant l'exécution de la requête.

Cette isolation snapshot (snapshot isolation) garantit que les lecteurs voient toujours un état cohérent de la table. Il n'y a jamais de lectures fantômes où des lignes apparaissent ou disparaissent au milieu d'une requête.

Concurrence Optimiste

Les écritures utilisent un protocole de concurrence optimiste. Plutôt que d'acquérir des verrous avant d'écrire, chaque écrivain suppose qu'il n'y aura pas de conflit et vérifie cette hypothèse au moment du commit.

Le processus de commit d'une écriture suit ces étapes :

1. **Lecture des métadonnées** — L'écrivain charge le fichier metadata.json courant et note le snapshot actif.
2. **Exécution de l'écriture** — L'écrivain crée les nouveaux fichiers de données, les fichiers manifestes et la liste de manifestes pour un nouveau snapshot.
3. **Création des métadonnées** — Un nouveau fichier metadata.json est préparé, référençant le nouveau snapshot.
4. **Validation du commit** — L'écrivain vérifie que le pointeur du catalogue n'a pas changé depuis la lecture initiale.

5. **Commit atomique** — Si le pointeur n'a pas changé, l'écrivain demande au catalogue de basculer atomiquement vers le nouveau fichier de métadonnées.
6. **Résolution de conflit** — Si le pointeur a changé (un autre écrivain a commité entre-temps), l'écrivain peut soit rejouer son opération sur le nouveau snapshot, soit abandonner.

Ce protocole évite les verrous coûteux tout en garantissant la cohérence. Les conflits sont détectés au moment du commit plutôt que prévenus par verrouillage préemptif.

Types de Conflits

Iceberg distingue plusieurs types de conflits avec des sémantiques de résolution différentes :

Conflits d'ajout — Deux écrivains ajoutent des données simultanément. Dans la plupart des cas, ces opérations sont compatibles et peuvent être fusionnées. L'écrivain qui détecte le conflit peut simplement rebaser son commit sur le nouveau snapshot.

Conflits de suppression — Un écrivain supprime des lignes qu'un autre écrivain modifie. Ce conflit nécessite généralement une intervention ou une logique de résolution spécifique à l'application.

Conflits de schéma — Un écrivain modifie le schéma pendant qu'un autre écrit des données. La résolution dépend de la compatibilité des changements de schéma avec les données écrites.

Le catalogue peut configurer le nombre de tentatives de résolution automatique avant d'abandonner :

```
ALTER TABLE events SET TBLPROPERTIES (
  'commit.retry.num-retries' = '10',
  'commit.retry.min-wait-ms' = '100',
  'commit.retry.max-wait-ms' = '60000'
);
```

Niveau d'Isolation Sérialisable

Pour les cas nécessitant des garanties plus fortes que l'isolation snapshot, Iceberg supporte l'isolation sérialisable. Ce mode garantit que les transactions s'exécutent comme si elles étaient séquentielles, même en présence de concurrence.

L'isolation sérialisable est obtenue en validant que les fichiers lus pendant la transaction n'ont pas été modifiés par d'autres transactions. Si une ligne lue a été modifiée, la transaction est abandonnée plutôt que de produire un résultat potentiellement incohérent.

```
-- Activation de l'isolation sérialisable
ALTER TABLE events SET TBLPROPERTIES (
  'write.wap.enabled' = 'true',
  'commit.retry.isolation-level' = 'serializable'
);
```

Les Tables de Métadonnées

Iceberg expose des tables virtuelles permettant d'interroger les métadonnées d'une table avec SQL standard. Ces tables de métadonnées sont essentielles pour le monitoring, le débogage et la compréhension de l'état d'une table.

Tables de Métadonnées Principales

snapshots — Liste tous les snapshots de la table avec leur timestamp, opération et statistiques de résumé.

```
SELECT snapshot_id, committed_at, operation, summary
FROM catalog.db.events.snapshots
ORDER BY committed_at DESC
LIMIT 10;
```

history — Affiche l'historique des snapshots courants au fil du temps, permettant de comprendre l'évolution de la table.

files — Énumère tous les fichiers de données du snapshot courant avec leurs statistiques.

```
SELECT file_path, record_count, file_size_in_bytes
FROM catalog.db.events.files
WHERE record_count > 1000000;
```

manifests — Liste les fichiers manifestes du snapshot courant.

partitions — Résume les partitions de la table avec des statistiques agrégées.

```
SELECT partition, record_count, file_count
FROM catalog.db.events.partitions
ORDER BY record_count DESC;
```

metadata_log_entries — Trace l'évolution des fichiers de métadonnées au fil du temps.

Utilisation pour le Diagnostic

Les tables de métadonnées sont précieuses pour diagnostiquer les problèmes de performance :

```
-- Identifier les petits fichiers nécessitant compaction
SELECT COUNT(*) as file_count,
       AVG(file_size_in_bytes) as avg_size,
       SUM(record_count) as total_records
FROM catalog.db.events.files
WHERE file_size_in_bytes < 10000000;

-- Vérifier l'accumulation de snapshots
SELECT COUNT(*) as snapshot_count,
       MIN(committed_at) as oldest_snapshot,
       MAX(committed_at) as newest_snapshot
FROM catalog.db.events.snapshots;

-- Analyser la distribution des partitions
SELECT partition, file_count, record_count
FROM catalog.db.events.partitions
WHERE file_count > 100
ORDER BY file_count DESC;
```

Ces requêtes permettent d'identifier les tables nécessitant maintenance avant que les problèmes n'impactent les performances de production.

Le Processus de Commit en Détail

Le processus de commit est le mécanisme central qui garantit les propriétés ACID des tables Iceberg. Comprendre ce processus en détail est essentiel pour diagnostiquer les problèmes de concurrence et optimiser les performances d'écriture.

Anatomie d'un Commit

Un commit Iceberg suit une séquence précise d'opérations qui transforme les données brutes en une mise à jour atomique de la table.

Phase 1 : Planification de l'écriture

L'écrivain détermine d'abord quelles données doivent être écrites et dans quels fichiers. Pour une insertion simple, cela implique de partitionner les données selon la spécification de partition et de regrouper les enregistrements par partition. Pour une mise à jour ou suppression, l'écrivain doit également identifier les fichiers existants affectés.

La configuration des tailles de fichiers cibles influence cette phase :

```
-- Configuration des tailles de fichiers
ALTER TABLE events SET TBLPROPERTIES (
  'write.target-file-size-bytes' = '536870912',  -- 512 Mo
  'write.parquet.row-group-size-bytes' = '134217728'  -- 128 Mo
);
```

Phase 2 : Écriture des fichiers de données

Les fichiers de données sont écrits dans le stockage objet. Chaque fichier est créé avec un nom unique (généralement un UUID) pour éviter les collisions. Les fichiers sont écrits complètement avant de passer à la phase suivante — il n'y a jamais de fichiers partiellement écrits référencés par les métadonnées.

Le format de fichier par défaut est Parquet avec compression Snappy, mais cela est configurable :

```
ALTER TABLE events SET TBLPROPERTIES (
  'write.format.default' = 'parquet',
  'write.parquet.compression-codec' = 'zstd',
  'write.parquet.compression-level' = '3'
);
```

Phase 3 : Création des manifestes

Une fois les fichiers de données écrits, l'écrivain crée les fichiers manifestes. Chaque manifeste est un fichier Avro qui liste les fichiers de données ajoutés (status = 1), supprimés (status = 2) ou inchangés (status = 0). Les statistiques de colonnes sont calculées et sérialisées dans le manifeste.

Pour les opérations d'append simples, un seul nouveau manifeste est généralement créé contenant tous les fichiers ajoutés. Pour les opérations de compaction ou de réécriture, les anciens manifestes peuvent être référencés avec les entrées marquées comme supprimées.

Phase 4 : Création de la liste de manifestes

La liste de manifestes est créée, référençant tous les manifestes du nouveau snapshot. Elle inclut les statistiques de partition agrégées pour chaque manifeste, permettant l'élagage au niveau manifeste.

Phase 5 : Préparation des métadonnées

Un nouveau fichier metadata.json est préparé. Il contient : - Le nouveau snapshot avec sa référence à la liste de manifestes - Le snapshot-log mis à jour - Potentiellement de nouvelles entrées dans schemas ou partition-specs si le schéma a évolué - Les propriétés de table mises à jour

Phase 6 : Commit atomique

C'est l'étape critique. L'écrivain demande au catalogue de basculer atomiquement le pointeur de métadonnées courantes vers le nouveau fichier. Le catalogue vérifie que le pointeur n'a pas changé depuis que l'écrivain a commencé, puis effectue la mise à jour.

Si le pointeur a changé (un autre écrivain a commité entre-temps), l'opération échoue et l'écrivain doit décider de rejouer ou d'abandonner.

Retry et Backoff Exponentiel

Lorsqu'un conflit est détecté, Iceberg implémente une stratégie de retry avec backoff exponentiel. À chaque tentative échouée, le temps d'attente augmente exponentiellement jusqu'au maximum configuré. Cette approche évite les tempêtes de retry où de nombreux écrivains réessaient simultanément, créant plus de conflits.

Les paramètres configurables incluent le nombre maximum de tentatives (typiquement 4), le temps d'attente minimum (100 ms), le temps d'attente maximum (60 secondes), et le temps total maximum de retry (30 minutes).

Gestion des Fichiers Orphelins

Un aspect critique du processus de commit est la gestion des fichiers orphelins. Si un écrivain crée des fichiers de données mais échoue avant le commit, ces fichiers restent dans le stockage sans être référencés par aucun snapshot.

Iceberg fournit des procédures pour nettoyer ces fichiers :

```
-- Suppression des fichiers orphelins plus vieux que 3 jours
CALL catalog.system.remove_orphan_files(
  table => 'db.events',
  older_than => TIMESTAMP '2024-01-01 00:00:00',
  dry_run => true
);
```

La procédure compare les fichiers présents dans le stockage avec ceux référencés dans les métadonnées et identifie les orphelins. Le mode dry_run permet de prévisualiser sans supprimer.

Branches et Tags

La spécification Iceberg inclut le support des branches et tags, apportant des concepts de contrôle de version similaires à Git directement aux tables de données.

Branches

Une branche est un pointeur nommé vers un snapshot qui peut évoluer indépendamment de la branche principale. Les cas d'usage incluent :

Développement isolé — Les ingénieurs peuvent créer une branche pour tester des transformations complexes sans affecter les données de production.

```
-- Création d'une branche de développement
ALTER TABLE events CREATE BRANCH dev;

-- Écriture sur la branche
INSERT INTO events.branch_dev SELECT ...;

-- Requête de la branche
SELECT * FROM events VERSION AS OF 'dev';
```

Pipelines de staging — Les données peuvent être écrites dans une branche de staging, validées, puis promues en production par fast-forward.

Expérimentation ML — Différentes versions des données d'entraînement peuvent être maintenues sur des branches séparées pour la reproductibilité des expériences.

Tags

Un tag est un pointeur nommé immutable vers un snapshot spécifique. Contrairement aux branches qui peuvent avancer, les tags sont permanents et représentent un moment précis de l'historique.

```
-- Création d'un tag pour marquer une release
ALTER TABLE events CREATE TAG end_of_q4_2024
  AS OF SNAPSHOT 8235603094578364387;

-- Requête du tag
SELECT * FROM events VERSION AS OF 'end_of_q4_2024';
```

Les tags sont utiles pour marquer les données utilisées pour entraîner un modèle en production, identifier les états de table correspondant aux rapports financiers, et créer des points de référence pour la reproduction des analyses.

Rétention des Branches et Tags

Les branches et tags sont indépendants de l'expiration des snapshots. Un snapshot référencé par un tag ou une branche n'est pas éligible à l'expiration, garantissant que les points de référence nommés restent accessibles.

```
-- Configuration de la rétention des branches
ALTER TABLE events CREATE BRANCH audit_2024
  RETAIN 365 DAYS
  WITH SNAPSHOT RETENTION 100 SNAPSHOTS;
```

Intégration avec les Moteurs de Requête

L'architecture ouverte d'Iceberg permet l'intégration avec de nombreux moteurs de requête. Chaque moteur interagit avec les métadonnées de manière similaire mais peut avoir des optimisations spécifiques.

Apache Spark

Spark est le moteur le plus mature pour Iceberg, avec un support complet des opérations DML et des procédures de maintenance.

```
# Configuration Spark pour Iceberg
spark = SparkSession.builder \
    .config("spark.sql.extensions",
            "org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions") \
    .config("spark.sql.catalog.catalog",
            "org.apache.iceberg.spark.SparkCatalog") \
    .config("spark.sql.catalog.catalog.type", "rest") \
    .config("spark.sql.catalog.catalog.uri", "http://catalog:8181") \
    .getOrCreate()
```

Spark exploite les statistiques de manifeste pour l'élagage de partition et les statistiques de fichier pour l'élagage de fichier. Le pushdown des prédicats est automatique pour les filtres sur les colonnes de partition et les colonnes avec statistiques.

Trino

Trino offre des performances de requête exceptionnelles sur Iceberg grâce à son architecture de mémoire distribuée et son optimiseur sophistiqué.

```
-- Création d'un catalogue Iceberg dans Trino
CREATE CATALOG iceberg WITH (
  connector.name = 'iceberg',
  iceberg.catalog.type = 'rest',
  iceberg.rest-catalog.uri = 'http://catalog:8181'
);
```

Trino est particulièrement efficace pour les requêtes ad-hoc et les jointures complexes. Il peut générer et utiliser les statistiques Puffin pour l'optimisation des jointures.

Apache Flink

Flink excelle pour l'ingestion streaming vers Iceberg, créant des commits réguliers à partir de flux de données continus.

Flink utilise le checkpointing pour coordonner les commits, garantissant la cohérence exactly-once même en cas de défaillance. Cette intégration permet de construire des architectures de Streaming Lakehouse où les données passent de Kafka à Iceberg en temps quasi-réel.

Dremio

Dremio est optimisé pour les requêtes analytiques sur les Lakehouses, avec des accélérations via la réflection et le caching. Dremio utilise les statistiques Iceberg pour l'optimisation mais maintient également ses propres statistiques supplémentaires via les réflections pour accélérer les requêtes récurrentes.

> **Étude de cas : Plateforme de Commerce Électronique**
> *Secteur* : Commerce de détail en ligne
> *Défi* : Unifier l'accès aux données entre Spark (ETL), Trino (analytics ad-hoc) et Flink (temps réel)

> *Solution* : Lakehouse Iceberg avec REST Catalog centralisé
> *Résultats* : Trois moteurs accèdent aux mêmes tables avec isolation complète, latence de requête ad-hoc réduite de 75 %, ingestion temps réel à 100 000 événements/seconde

Considérations Avancées de Performance

La compréhension de l'anatomie technique d'Iceberg permet d'anticiper et de résoudre les problèmes de performance. Plusieurs facteurs influencent les performances des tables Iceberg et méritent une attention particulière.

Prolifération de Petits Fichiers

Le streaming et les écritures fréquentes peuvent créer de nombreux petits fichiers, dégradant les performances de lecture. Chaque fichier implique un overhead de lecture des métadonnées et d'ouverture de fichier. Une table avec un million de fichiers de 1 Mo sera significativement plus lente à interroger qu'une table avec mille fichiers de 1 Go.

Les causes de prolifération incluent : - Commits fréquents en streaming sans batching adéquat - Partitionnement trop fin créant de nombreuses partitions avec peu de données - Opérations de mise à jour fréquentes en mode COW - Manque de compaction régulière

Solutions recommandées :

```
-- Configurer une taille cible de fichier appropriée
ALTER TABLE events SET TBLPROPERTIES (
  'write.target-file-size-bytes' = '536870912'  -- 512 Mo
);

-- Exécuter une compaction pour fusionner les petits fichiers
CALL catalog.system.rewrite_data_files(
  table => 'db.events',
  strategy => 'binpack',
  options => map('target-file-size-bytes', '536870912')
);
```

La compaction binpack fusionne les petits fichiers jusqu'à atteindre la taille cible. Des stratégies plus sophistiquées comme sort permettent également de trier les données pour améliorer la localité.

Accumulation de Snapshots

Sans expiration, les snapshots s'accumulent indéfiniment, gonflant les métadonnées et les coûts de stockage. Chaque snapshot référence tous les fichiers de la table à ce moment, donc même si les données changent peu, les fichiers de métadonnées grossissent.

Solutions recommandées :

```
-- Configuration de l'expiration automatique
ALTER TABLE events SET TBLPROPERTIES (
  'history.expire.max-snapshot-age-ms' = '604800000',  -- 7 jours
  'history.expire.min-snapshots-to-keep' = '10'
);
```

```
-- Expiration manuelle des anciens snapshots
CALL catalog.system.expire_snapshots(
  table => 'db.events',
  older_than => TIMESTAMP '2024-01-01 00:00:00',
  retain_last => 10
);
```

L'expiration supprime les snapshots obsolètes et les fichiers de données qui ne sont plus référencés par aucun snapshot restant. Cette opération doit être planifiée régulièrement pour maintenir la santé de la table.

Fichiers de Suppression Non Compactés

En mode Merge-on-Read, les fichiers de suppression s'accumulent et dégradent les performances de lecture car chaque lecture doit appliquer les suppressions. Une table avec des milliers de fichiers de suppression peut voir ses temps de requête multiplier par 10 ou plus.

Indicateurs de problème : - Ratio élevé fichiers de suppression / fichiers de données - Temps de planification de requête anormalement long - Performance de lecture dégradée sans augmentation de volume

Solutions recommandées :

```
-- Compaction majeure pour absorber les suppressions
CALL catalog.system.rewrite_data_files(
  table => 'db.events',
  strategy => 'sort',
  options => map(
    'target-file-size-bytes', '536870912',
    'rewrite-all', 'true'
  )
);

-- Réécriture spécifique des fichiers de position delete
CALL catalog.system.rewrite_position_delete_files(
  table => 'db.events'
);
```

Statistiques Obsolètes

Des statistiques obsolètes peuvent conduire à des plans de requête sous-optimaux, particulièrement pour les jointures. L'optimiseur basé sur les coûts (CBO) utilise les statistiques NDV pour estimer les cardinalités et choisir les stratégies de jointure.

Solutions recommandées :

```
-- Régénérer les statistiques après modifications majeures
CALL catalog.system.compute_table_stats(
  table => 'db.events',
  columns => array('user_id', 'product_id', 'event_date')
);
```

Taille du Fichier de Métadonnées

Le fichier metadata.json peut devenir volumineux si l'historique complet des snapshots et des schémas y est conservé. Un fichier de métadonnées de plusieurs centaines de mégaoctets ralentit chaque accès à la table.

Solutions recommandées :

```
-- Limiter l'historique conservé dans les métadonnées
ALTER TABLE events SET TBLPROPERTIES (
  'write.metadata.delete-after-commit.enabled' = 'true',
  'write.metadata.previous-versions-max' = '100'
);
```

Monitoring de la Santé des Tables

Le monitoring proactif de la santé des tables permet d'identifier les problèmes avant qu'ils n'impactent la production.

```
-- Tableau de bord de santé via les tables de métadonnées
SELECT
  'file_count' as metric,
  COUNT(*) as value,
  CASE
    WHEN COUNT(*) > 10000 THEN 'WARNING: Consider compaction'
    ELSE 'OK'
  END as status
FROM catalog.db.events.files
UNION ALL
SELECT
  'avg_file_size_mb' as metric,
  AVG(file_size_in_bytes) / 1048576 as value,
  CASE
    WHEN AVG(file_size_in_bytes) < 10485760 THEN 'WARNING: Small files detected'
    ELSE 'OK'
  END as status
FROM catalog.db.events.files
UNION ALL
SELECT
  'snapshot_count' as metric,
  COUNT(*) as value,
  CASE
    WHEN COUNT(*) > 1000 THEN 'WARNING: Consider snapshot expiration'
    ELSE 'OK'
  END as status
FROM catalog.db.events.snapshots;
```

Ce type de monitoring peut être automatisé et intégré aux systèmes d'alerting existants pour maintenir la santé des tables Iceberg en production.

Résumé

Ce chapitre a exploré en profondeur l'anatomie technique d'Apache Iceberg, révélant les mécanismes qui permettent au format de table de délivrer des performances et une fiabilité de niveau entreprise. La

compréhension de ces composants internes est essentielle pour tout architecte ou ingénieur de données travaillant avec le Data Lakehouse moderne.

Architecture en Trois Couches

L'architecture d'Iceberg sépare clairement les préoccupations entre le catalogue, les métadonnées et les données. Le catalogue maintient les références aux tables et garantit l'atomicité des commits. La couche de métadonnées, organisée hiérarchiquement en fichiers metadata.json, listes de manifestes et manifestes, fournit toute l'information nécessaire pour reconstruire l'état de la table à n'importe quel moment. La couche de données stocke les enregistrements dans des formats optimisés comme Parquet.

Cette séparation permet une flexibilité maximale : les moteurs de requête peuvent être changés sans modifier les données, les catalogues peuvent être migrés, et les stratégies de stockage peuvent évoluer. L'isolation des lectures garantit que les requêtes ne sont jamais affectées par les écritures concurrentes.

Hiérarchie de Métadonnées

La hiérarchie de métadonnées — metadata.json vers manifest list vers manifest vers data files — forme une structure arborescente permettant un accès efficace à différents niveaux de granularité. Cette architecture permet l'élagage progressif : d'abord au niveau des manifestes via les statistiques de partition, puis au niveau des fichiers via les bornes de colonnes, et finalement au niveau des groupes de lignes dans les fichiers Parquet.

Les statistiques collectées à chaque niveau — comptages, bornes min/max, valeurs null — alimentent l'optimiseur pour éliminer les données non pertinentes avant même de les lire. Pour les requêtes sélectives, cette cascade d'élagage peut réduire le volume lu de plusieurs ordres de grandeur.

Système d'Identifiants de Colonnes

Le système d'identifiants uniques permanents pour chaque champ est l'innovation qui rend possible l'évolution de schéma sans effets de bord. Contrairement aux systèmes basés sur la position ou le nom, l'identification par ID permet de renommer, réordonner et supprimer des colonnes sans impact sur les fichiers existants.

Les colonnes ajoutées reçoivent automatiquement de nouveaux identifiants, garantissant qu'aucune donnée historique ne sera accidentellement interprétée avec le nouveau schéma. Cette approche permet aux tables de longue durée d'évoluer naturellement avec les besoins métier sans migrations coûteuses.

Partitionnement Masqué et Évolution

Le partitionnement masqué libère les utilisateurs de la gestion explicite des partitions. Les transformations — year, month, day, hour, bucket, truncate — dérivent automatiquement les valeurs de partition des colonnes source. Les producteurs n'ont pas besoin de connaître le schéma de partitionnement, et les consommateurs écrivent des requêtes naturelles.

L'évolution du partitionnement permet de changer le schéma sans réécrire les données existantes. Les anciennes et nouvelles spécifications coexistent, avec une planification de requête appropriée pour chaque layout. Cette flexibilité est cruciale pour adapter les tables aux volumes croissants et aux patterns d'accès évolutifs.

Stratégies de Modification au Niveau Lignes

Les stratégies Copy-on-Write et Merge-on-Read offrent des compromis configurables entre performance d'écriture et de lecture. COW réécrit les fichiers affectés pour une lecture optimale, tandis que MOR crée des fichiers de suppression pour une écriture rapide.

Les fichiers de suppression positionnelle et d'égalité permettent différents niveaux de compromis. La version 3 introduit les vecteurs de suppression pour une représentation plus efficace. La possibilité de configurer indépendamment DELETE, UPDATE et MERGE permet une optimisation fine selon les patterns d'utilisation.

Statistiques Multi-Niveaux

Le système de statistiques d'Iceberg opère à plusieurs niveaux. Les manifestes stockent les comptages et bornes de colonnes. Les filtres de Bloom accélèrent les recherches ponctuelles. Les fichiers Puffin stockent les statistiques avancées comme le NDV via les sketches Theta.

Ces statistiques alimentent l'optimisation à chaque étape : planification des requêtes, choix des stratégies de jointure, allocation des ressources. L'investissement dans la collecte et la maintenance des statistiques se traduit directement en performances de requête améliorées.

Concurrence Optimiste et Transactions

Le protocole de concurrence optimiste permet une haute concurrence sans verrous coûteux. Les écrivains préparent leurs modifications indépendamment et valident au moment du commit. Les conflits sont détectés et peuvent être résolus automatiquement par retry avec backoff exponentiel.

Les garanties ACID — atomicité, cohérence, isolation, durabilité — sont maintenues par ce protocole. Les lecteurs bénéficient de l'isolation snapshot, voyant toujours un état cohérent même pendant les écritures concurrentes.

Branches et Tags

Le support des branches et tags apporte le contrôle de version Git aux tables de données. Les branches permettent le développement isolé et les pipelines de staging. Les tags marquent les états importants pour l'audit et la reproductibilité.

Ces fonctionnalités sont particulièrement précieuses pour le Machine Learning (versionnement des données d'entraînement), la conformité (audit trails) et les opérations (rollback facile).

Implications Pratiques

Pour l'architecte de données, la compréhension de ces mécanismes guide les décisions de conception. Le choix entre COW et MOR dépend du ratio lecture/écriture. Le schéma de partitionnement doit correspondre aux patterns de requête. La configuration de l'expiration des snapshots équilibre le voyage dans le temps et les coûts de stockage.

Pour l'ingénieur de données, cette connaissance permet le diagnostic efficace des problèmes. Les tables de métadonnées révèlent l'accumulation de petits fichiers, les fichiers de suppression non compactés, et les snapshots en croissance. Les procédures de maintenance — compaction, expiration, nettoyage d'orphelins — résolvent ces problèmes.

Pour l'équipe d'opérations, les détails techniques éclairent les procédures. Le monitoring de la santé des tables prévient les dégradations de performance. La planification des maintenances minimise l'impact sur les charges de production. La compréhension des conflits guide la configuration de la concurrence.

Perspectives Techniques

L'évolution continue de la spécification Iceberg apporte régulièrement de nouvelles capacités. La version 3 ajoute les vecteurs de suppression, les types variant pour les données semi-structurées, et les types géospatiaux. La version 4, en développement, promet de nouvelles optimisations et fonctionnalités.

L'ajout prévu d'un endpoint de scan planning au REST Catalog permettra aux catalogues d'optimiser et de cacher les plans de requête. Les vues interopérables standardiseront la définition des vues entre moteurs. Ces évolutions renforceront le positionnement d'Iceberg comme standard universel du Data Lakehouse.

Le chapitre suivant appliquera ces connaissances techniques à des scénarios concrets d'implémentation, démontrant comment créer, alimenter et interroger des tables Iceberg avec différents moteurs de requête dans un environnement de production.

Chapitre IV.3 — Mise en Pratique avec Apache Iceberg

Les deux chapitres précédents ont établi les fondations conceptuelles du Data Lakehouse et l'anatomie technique d'Apache Iceberg. Nous avons exploré la hiérarchie des métadonnées, les stratégies d'écriture Copy-on-Write et Merge-on-Read, l'évolution de schéma et le partitionnement masqué. Ces connaissances théoriques constituent un socle essentiel, mais la véritable maîtrise d'une technologie ne s'acquiert qu'à travers la pratique.

Ce chapitre marque le passage de la théorie à l'implémentation concrète. Nous allons construire, étape par étape, un environnement Lakehouse fonctionnel sur votre machine locale, créer des tables Iceberg avec Apache Spark, les interroger via Dremio, puis connecter l'ensemble à un outil d'intelligence d'affaires (BI) pour produire des visualisations exploitables. Cette progression illustre le parcours complet des données dans une architecture Lakehouse moderne : de l'ingestion brute jusqu'à la consommation analytique.

L'objectif n'est pas de présenter une configuration de production — les chapitres de la Partie 2 couvriront ces aspects en profondeur — mais de démystifier Apache Iceberg à travers une expérience pratique immersive. En suivant les exercices de ce chapitre, vous développerez une intuition concrète du fonctionnement d'Iceberg, de ses interactions avec les différents moteurs de calcul et de sa valeur ajoutée pour les analystes et les scientifiques de données.

L'approche retenue privilégie la conteneurisation avec Docker, permettant un déploiement reproductible et isolé de l'environnement de développement. Cette méthode présente plusieurs avantages significatifs pour l'apprentissage : elle élimine les problèmes de compatibilité entre systèmes d'exploitation, garantit que chaque lecteur dispose d'une configuration identique, et facilite le nettoyage complet de l'environnement après les expérimentations. De plus, les patrons architecturaux établis dans cet environnement local se transposent directement vers des déploiements infonuagiques de production.

Le parcours pratique de ce chapitre s'articule autour d'un scénario réaliste : la construction d'une plateforme analytique pour une entreprise de commerce électronique canadienne. Ce fil conducteur nous permettra d'explorer la création de tables dimensionnelles (clients) et factuelles (transactions), les jointures entre tables, les agrégations analytiques et la visualisation des indicateurs clés de performance. Les données d'exemple utilisent des noms et des provinces canadiennes, rendant les résultats plus tangibles pour le public cible de cette monographie.

IV.3.1 Configuration d'un environnement Apache Iceberg

La mise en place d'un environnement Lakehouse complet nécessitait traditionnellement des semaines de configuration et une infrastructure coûteuse. Grâce à la conteneurisation et aux images Docker préconfigurées, nous pouvons désormais déployer un écosystème complet en quelques minutes sur un ordinateur portable standard. Cette démocratisation de l'accès aux technologies Lakehouse représente une avancée majeure pour l'apprentissage et le prototypage rapide.

L'écosystème que nous allons déployer reproduit fidèlement les composants d'une architecture Lakehouse de production, mais à une échelle adaptée aux ressources d'une machine de développement. Chaque service remplit une fonction spécifique et communique avec les autres via des interfaces standardisées, illustrant le principe de découplage qui caractérise les architectures modernes de données.

Prérequis techniques

Avant de commencer, assurez-vous de disposer des éléments suivants sur votre machine de développement :

Composant	Version minimale	Recommandation
Docker Desktop	4.25+	Dernière version stable
Docker Compose	2.20+	Inclus dans Docker Desktop
Mémoire RAM	8 Go	16 Go recommandés
Espace disque	10 Go libres	20 Go pour expérimentation
Ports disponibles	8080, 8888, 9000, 9001, 9047, 19120	Vérifier avec `netstat` ou `lsof`

Pour les utilisateurs de systèmes GNU/Linux ou macOS, la vérification de la disponibilité des ports s'effectue avec la commande suivante :

```
# Vérifier si un port est disponible
lsof -i :9000 || echo "Port 9000 disponible"
```

Architecture de l'environnement de développement

Notre environnement de développement reproduit les couches fondamentales d'un Lakehouse en production, mais à une échelle adaptée à l'apprentissage. Chaque composant joue un rôle spécifique dans l'architecture globale :

MinIO sert de couche de stockage compatible S3. Cette solution de stockage objet à code source ouvert émule parfaitement l'interface programmatique (API) d'Amazon S3, permettant de développer localement des applications qui fonctionneront ensuite sur n'importe quel stockage infonuagique compatible S3. MinIO offre une performance remarquable et une simplicité de configuration idéale pour le développement.

Project Nessie assume le rôle de catalogue Iceberg. Ce catalogue REST respecte la spécification Apache Iceberg REST Catalog et ajoute des capacités de versionnement de type Git pour les données. Nessie permet de créer des branches, de fusionner des modifications et de maintenir un historique complet des changements sur les tables Iceberg.

Apache Spark constitue le moteur de traitement principal. Spark offre le support le plus complet pour Apache Iceberg, incluant toutes les opérations DDL (Data Definition Language) et DML (Data Manipulation Language). La version 3.5 apporte des améliorations significatives pour les opérations MERGE et la gestion des métadonnées.

Dremio représente la couche de fédération et d'accélération. Cette plateforme Lakehouse unifie l'accès aux données provenant de sources multiples et optimise les performances des requêtes grâce à ses réflexions de données (data reflections) et son moteur de requêtes basé sur Apache Arrow.

Apache Superset (optionnel) fournit les capacités de visualisation. Cet outil d'intelligence d'affaires à code source ouvert permet de créer des tableaux de bord interactifs directement connectés à Dremio ou aux tables Iceberg.

Fichier Docker Compose

Créez un fichier nommé `docker-compose.yml` dans un répertoire dédié à votre projet Lakehouse :

```yaml
version: '3.8'

services:
  # Stockage objet compatible S3
  minio:
    image: minio/minio:latest
    container_name: minio
    environment:
      MINIO_ROOT_USER: admin
      MINIO_ROOT_PASSWORD: password123
      MINIO_REGION: ca-central-1
    ports:
      - "9000:9000"
      - "9001:9001"
    command: server /data --console-address ":9001"
    volumes:
      - minio-data:/data
    networks:
      - lakehouse-network
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:9000/minio/health/live"]
      interval: 30s
      timeout: 20s
      retries: 3

  # Catalogue Iceberg REST avec versionnement
  nessie:
    image: projectnessie/nessie:latest
    container_name: nessie
    environment:
      QUARKUS_PROFILE: prod
      QUARKUS_HTTP_PORT: 19120
      NESSIE_VERSION_STORE_TYPE: IN_MEMORY
    ports:
      - "19120:19120"
    networks:
      - lakehouse-network
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:19120/api/v2/config"]
      interval: 30s
      timeout: 10s
      retries: 3

  # Moteur de traitement Spark avec Iceberg
  spark-iceberg:
    image: tabulario/spark-iceberg:3.5.1_1.5.2
    container_name: spark-iceberg
    depends_on:
      - minio
      - nessie
```

```
    environment:
      AWS_ACCESS_KEY_ID: admin
      AWS_SECRET_ACCESS_KEY: password123
      AWS_REGION: ca-central-1
      SPARK_HOME: /opt/spark
    ports:
      - "8888:8888"
      - "8080:8080"
      - "10000:10000"
      - "10001:10001"
    volumes:
      - spark-warehouse:/home/iceberg/warehouse
      - ./notebooks:/home/iceberg/notebooks
    networks:
      - lakehouse-network
    command: notebook

  # Plateforme Lakehouse Dremio
  dremio:
    image: dremio/dremio-oss:latest
    container_name: dremio
    depends_on:
      - minio
      - nessie
    ports:
      - "9047:9047"
      - "31010:31010"
      - "32010:32010"
      - "45678:45678"
    volumes:
      - dremio-data:/opt/dremio/data
    networks:
      - lakehouse-network

volumes:
  minio-data:
  spark-warehouse:
  dremio-data:

networks:
  lakehouse-network:
    driver: bridge
```

Initialisation de l'environnement

Lancez l'ensemble des services avec la commande suivante :

```
# Créer le répertoire pour les notebooks
mkdir -p notebooks

# Démarrer l'environnement
docker-compose up -d

# Vérifier le statut des conteneurs
docker-compose ps
```

L'initialisation complète prend généralement entre deux et cinq minutes selon la performance de votre machine et la disponibilité des images dans le cache Docker local.

Configuration du stockage MinIO

Avant de créer des tables Iceberg, nous devons préparer l'infrastructure de stockage dans MinIO. Accédez à la console MinIO via votre navigateur à l'adresse `http://localhost:9001` et connectez-vous avec les identifiants définis dans le fichier Docker Compose (`admin` / `password123`).

Créez un compartiment (bucket) nommé `warehouse` qui servira d'entrepôt principal pour les données et métadonnées Iceberg. Ce compartiment contiendra la structure hiérarchique suivante :

```
warehouse/
├── db_production/
│   ├── clients/
│   │   ├── data/
│   │   └── metadata/
│   └── transactions/
│       ├── data/
│       └── metadata/
└── db_staging/
    └── ...
```

Vous pouvez également créer le compartiment programmatiquement via l'interface en ligne de commande MinIO :

```
# Installer le client MinIO (mc)
docker exec -it minio mc alias set local http://localhost:9000 admin password123

# Créer le compartiment principal
docker exec -it minio mc mb local/warehouse

# Vérifier la création
docker exec -it minio mc ls local/
```

Validation de la connectivité

Avant de poursuivre, validez que tous les composants sont opérationnels et peuvent communiquer entre eux :

Service	URL de vérification	État attendu
MinIO Console	http://localhost:9001	Interface de connexion
MinIO API	http://localhost:9000/minio/health/live	`OK`
Nessie	http://localhost:19120/api/v2/config	Configuration JSON
Spark UI	http://localhost:8080	Interface Spark
Jupyter	http://localhost:8888	Interface de notebooks
Dremio	http://localhost:9047	Assistant de configuration

Un script de validation automatisé peut simplifier cette vérification :

```
#!/bin/bash
# validation_lakehouse.sh

echo "=== Validation de l'environnement Lakehouse ==="

services=("minio:9000" "nessie:19120" "spark-iceberg:8888" "dremio:9047")
for service in "${services[@]}"; do
    IFS=':' read -r name port <<< "$service"
    if curl -s --connect-timeout 5 "http://localhost:$port" > /dev/null 2>&1; then
        echo "✓ $name (port $port) : Opérationnel"
    else
        echo "✗ $name (port $port) : Non accessible"
    fi
done

echo "=== Validation terminée ==="
```

Résolution des problèmes courants

L'expérience montre que certains problèmes surviennent fréquemment lors de la configuration initiale. Voici les solutions aux difficultés les plus courantes :

Problème : Conteneurs qui redémarrent en boucle

Ce comportement indique généralement une insuffisance de ressources ou un conflit de ports. Vérifiez les journaux du conteneur concerné :

```
docker logs spark-iceberg --tail 50
```

Solutions possibles : - Augmentez la mémoire allouée à Docker Desktop (Settings → Resources) - Libérez les ports utilisés par d'autres applications - Supprimez les volumes corrompus : `docker-compose down -v && docker-compose up -d`

Problème : Erreur de connexion S3 dans Spark

Les messages d'erreur contenant `Unable to execute HTTP request` ou `Access Denied` indiquent un problème de configuration des identifiants ou de l'endpoint S3.

Vérifiez : - La cohérence des identifiants entre le fichier Docker Compose et la configuration Spark - L'accessibilité du service MinIO depuis le conteneur Spark : `docker exec spark-iceberg curl http://minio:9000` - L'activation de l'accès par chemin (path-style) : `fs.s3a.path.style.access=true`

Problème : Nessie ne répond pas aux requêtes

Le catalogue Nessie peut mettre jusqu'à 30 secondes à s'initialiser complètement. Si les problèmes persistent :

```
# Vérifier les journaux Nessie
docker logs nessie

# Redémarrer le service
docker-compose restart nessie
```

> **Performance**
> Sur une machine équipée de 16 Go de RAM, allouez au moins 8 Go à Docker Desktop pour garantir des performances optimales. Les opérations Spark sur des jeux de données volumineux peuvent être limitées par la mémoire disponible dans l'environnement conteneurisé. Pour des charges de travail plus importantes, envisagez d'utiliser un cluster Spark distant ou des services infonuagiques gérés comme Amazon EMR, Google Dataproc ou Azure HDInsight.

Comprendre l'architecture des fichiers Iceberg sur MinIO

Avant de passer à la création de tables, il est instructif de comprendre comment Apache Iceberg organise les données sur le stockage objet. Lorsque vous créerez vos premières tables, MinIO contiendra la structure suivante :

```
warehouse/
├── ecommerce/                          # Espace de noms (namespace)
│   ├── clients/                        # Répertoire de la table
│   │   ├── metadata/                   # Métadonnées Iceberg
│   │   │   ├── v1.metadata.json        # Fichier de métadonnées (version 1)
│   │   │   ├── v2.metadata.json        # Fichier de métadonnées (version 2)
│   │   │   ├── snap-1234567890.avro    # Fichier manifest list
│   │   │   └── 0000-abc123.avro        # Fichier manifest
│   │   └── data/                       # Fichiers de données
│   │       ├── date_inscription_month=2024-01/
│   │       │   └── province=QC/
│   │       │       └── 00000-0-abc.parquet
│   │       └── date_inscription_month=2024-02/
│   │           └── ...
│   └── transactions/
│       ├── metadata/
│       └── data/
```

Cette organisation reflète les concepts abordés au Chapitre IV.2 :

- **Fichiers de métadonnées** (`*.metadata.json`) : Pointent vers les manifest lists et contiennent le schéma, les propriétés de table et l'historique des snapshots.
- **Manifest lists** (`snap-*.avro`) : Répertorient les fichiers manifest associés à chaque snapshot.
- **Manifests** (`0000-*.avro`) : Cataloguent les fichiers de données avec leurs statistiques (min/max, nombre d'enregistrements).
- **Fichiers de données** (`*.parquet`) : Contiennent les données réelles au format Parquet, organisées par partition.

IV.3.2 Création de tables Iceberg dans Spark

Apache Spark demeure le moteur de traitement de référence pour Apache Iceberg, offrant le support le plus complet des fonctionnalités du format de table. Dans cette section, nous allons créer notre première table Iceberg, y insérer des données, explorer les capacités de voyage dans le temps (time travel) et démontrer l'évolution de schéma.

Accès à l'environnement Spark

Ouvrez l'interface Jupyter Notebook en naviguant vers `http://localhost:8888`. L'image Docker `tabulario/spark-iceberg` inclut une configuration Spark préconfigurée pour Iceberg avec un catalogue Nessie. Créez un nouveau notebook Python et exécutez les cellules suivantes pour initialiser votre session Spark.

```
from pyspark.sql import SparkSession
from pyspark.sql.types import (
    StructType, StructField, StringType,
    IntegerType, DoubleType, TimestampType, DateType
)
from datetime import datetime, date
import os

# Configuration de la session Spark avec Iceberg et Nessie
spark = SparkSession.builder \
    .appName("Lakehouse-Demo") \
    .config("spark.jars.packages",
            "org.apache.iceberg:iceberg-spark-runtime-3.5_2.12:1.5.2,"
            "org.projectnessie.nessie-integrations:nessie-spark-
extensions-3.5_2.12:0.77.1,"
            "software.amazon.awssdk:bundle:2.24.8,"
            "software.amazon.awssdk:url-connection-client:2.24.8") \
    .config("spark.sql.extensions",
            "org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions,"
            "org.projectnessie.spark.extensions.NessieSparkSessionExtensions") \
    .config("spark.sql.catalog.nessie", "org.apache.iceberg.spark.SparkCatalog") \
    .config("spark.sql.catalog.nessie.uri", "http://nessie:19120/api/v2") \
    .config("spark.sql.catalog.nessie.ref", "main") \
    .config("spark.sql.catalog.nessie.authentication.type", "NONE") \
    .config("spark.sql.catalog.nessie.catalog-impl",
            "org.apache.iceberg.nessie.NessieCatalog") \
    .config("spark.sql.catalog.nessie.warehouse", "s3://warehouse") \
    .config("spark.sql.catalog.nessie.io-impl",
            "org.apache.iceberg.aws.s3.S3FileIO") \
    .config("spark.sql.catalog.nessie.s3.endpoint", "http://minio:9000") \
    .config("spark.sql.catalog.nessie.s3.path-style-access", "true") \
    .config("spark.hadoop.fs.s3a.endpoint", "http://minio:9000") \
    .config("spark.hadoop.fs.s3a.access.key", "admin") \
    .config("spark.hadoop.fs.s3a.secret.key", "password123") \
    .config("spark.hadoop.fs.s3a.path.style.access", "true") \
    .config("spark.hadoop.fs.s3a.impl",
            "org.apache.hadoop.fs.s3a.S3AFileSystem") \
    .getOrCreate()

print(f"Session Spark initialisée : {spark.version}")
```

Création de l'espace de noms et de la table

Dans Apache Iceberg, les tables sont organisées dans des espaces de noms (namespaces) qui correspondent conceptuellement aux bases de données dans un système de gestion de base de données relationnelle (SGBDR) traditionnel. Créons un espace de noms pour notre démonstration :

```
# Créer l'espace de noms (équivalent d'une base de données)
spark.sql("CREATE NAMESPACE IF NOT EXISTS nessie.ecommerce")
```

```
# Vérifier les espaces de noms disponibles
spark.sql("SHOW NAMESPACES IN nessie").show()
```

Maintenant, créons une table `clients` avec un schéma représentatif d'une application de commerce électronique. Cette table utilisera le partitionnement masqué (hidden partitioning) d'Iceberg sur la colonne de date d'inscription :

```
# Création de la table clients avec partitionnement par mois
spark.sql("""
    CREATE TABLE IF NOT EXISTS nessie.ecommerce.clients (
        client_id BIGINT,
        prenom STRING,
        nom STRING,
        courriel STRING,
        province STRING,
        date_inscription DATE,
        segment STRING,
        valeur_vie DOUBLE
    )
    USING iceberg
    PARTITIONED BY (months(date_inscription), province)
    TBLPROPERTIES (
        'write.format.default' = 'parquet',
        'write.parquet.compression-codec' = 'zstd',
        'commit.retry.num-retries' = '4',
        'write.metadata.delete-after-commit.enabled' = 'true',
        'write.metadata.previous-versions-max' = '100'
    )
""")

print("Table 'clients' créée avec succès")
```

Examinons les propriétés importantes de cette définition :

- `PARTITIONED BY (months(date_inscription), province)` : Le partitionnement masqué transforme automatiquement la date d'inscription en partitions mensuelles sans exposer cette complexité aux utilisateurs finaux. Les requêtes peuvent filtrer directement sur `date_inscription` sans connaître la structure de partitionnement.
- `write.format.default = 'parquet'` : Les données sont stockées au format Parquet, optimisé pour les requêtes analytiques avec sa structure colonnaire.
- `write.parquet.compression-codec = 'zstd'` : L'algorithme de compression Zstandard offre un excellent compromis entre taux de compression et vitesse de décompression.
- `commit.retry.num-retries = 4` : Cette propriété définit le nombre de tentatives en cas de conflit d'écriture concurrente, assurant la robustesse des opérations transactionnelles.
- `write.metadata.delete-after-commit.enabled = true` : Active le nettoyage automatique des anciens fichiers de métadonnées après chaque commit, évitant l'accumulation de fichiers obsolètes.
- `write.metadata.previous-versions-max = 100` : Limite le nombre de versions de métadonnées conservées, équilibrant la capacité de voyage dans le temps avec l'efficacité du stockage.

Ces propriétés illustrent la richesse de configuration offerte par Apache Iceberg. Dans un environnement de production, ces paramètres seraient ajustés en fonction des patrons d'accès aux données, des exigences

de conformité et des contraintes de performance spécifiques à chaque cas d'utilisation. Le Chapitre IV.10 approfondit les stratégies d'optimisation et de maintenance des tables Iceberg.

Insertion de données

Insérons un ensemble de données représentatif de clients canadiens :

```
from pyspark.sql import Row

# Données d'exemple représentant des clients canadiens
donnees_clients = [
    Row(1001, "Marie", "Tremblay", "marie.tremblay@courriel.ca", "QC",
        date(2024, 1, 15), "Premium", 12500.00),
    Row(1002, "Jean", "Gagnon", "jean.gagnon@courriel.ca", "QC",
        date(2024, 1, 22), "Standard", 3200.00),
    Row(1003, "Sarah", "Chen", "sarah.chen@email.ca", "ON",
        date(2024, 2, 3), "Premium", 18900.00),
    Row(1004, "Mohammed", "Hassan", "m.hassan@mail.ca", "ON",
        date(2024, 2, 14), "Standard", 2100.00),
    Row(1005, "Emily", "Smith", "emily.smith@email.ca", "BC",
        date(2024, 2, 28), "Enterprise", 45000.00),
    Row(1006, "Pierre", "Lavoie", "p.lavoie@courriel.ca", "QC",
        date(2024, 3, 5), "Premium", 8700.00),
    Row(1007, "Anika", "Patel", "anika.patel@email.ca", "AB",
        date(2024, 3, 12), "Standard", 4300.00),
    Row(1008, "François", "Dubois", "f.dubois@courriel.ca", "QC",
        date(2024, 3, 20), "Premium", 15200.00),
    Row(1009, "Wei", "Zhang", "wei.zhang@email.ca", "BC",
        date(2024, 4, 1), "Enterprise", 52000.00),
    Row(1010, "Sophie", "Martin", "sophie.martin@courriel.ca", "QC",
        date(2024, 4, 8), "Standard", 2800.00),
]

# Définir le schéma explicitement
schema_clients = StructType([
    StructField("client_id", IntegerType(), False),
    StructField("prenom", StringType(), True),
    StructField("nom", StringType(), True),
    StructField("courriel", StringType(), True),
    StructField("province", StringType(), True),
    StructField("date_inscription", DateType(), True),
    StructField("segment", StringType(), True),
    StructField("valeur_vie", DoubleType(), True),
])

# Créer le DataFrame et insérer les données
df_clients = spark.createDataFrame(donnees_clients, schema_clients)
df_clients.writeTo("nessie.ecommerce.clients").append()

print(f"Insertion de {df_clients.count()} enregistrements réussie")
```

Validation et exploration des données

Vérifions que les données ont été correctement insérées et explorons la structure créée par Iceberg :

```
# Requête de validation
spark.sql("""
    SELECT province, segment, COUNT(*) as nb_clients,
           ROUND(AVG(valeur_vie), 2) as valeur_moyenne
    FROM nessie.ecommerce.clients
    GROUP BY province, segment
    ORDER BY province, segment
""").show()

# Examiner les métadonnées de la table
spark.sql("DESCRIBE EXTENDED nessie.ecommerce.clients").show(truncate=False)
```

Exploration des snapshots et du voyage dans le temps

L'une des fonctionnalités les plus puissantes d'Apache Iceberg est sa capacité de voyage dans le temps (time travel). Chaque opération d'écriture crée un nouveau snapshot, permettant de consulter l'état de la table à n'importe quel moment dans le passé. Cette fonctionnalité s'avère particulièrement précieuse pour plusieurs cas d'utilisation :

- **Audit et conformité** : Reconstituer l'état exact des données à un moment donné pour répondre aux exigences réglementaires
- **Débogage** : Comparer les données avant et après une transformation pour identifier les anomalies
- **Récupération** : Restaurer une table à un état antérieur après une modification erronée
- **Analyse temporelle** : Comparer les métriques entre différentes périodes avec une cohérence garantie

```
# Consulter l'historique des snapshots
spark.sql("""
    SELECT
        snapshot_id,
        committed_at,
        operation,
        summary['added-records'] as enregistrements_ajoutes,
        summary['total-records'] as total_enregistrements
    FROM nessie.ecommerce.clients.snapshots
    ORDER BY committed_at DESC
""").show(truncate=False)
```

Cette requête révèle l'historique complet des modifications de la table. Le champ `operation` indique le type d'opération (append, overwrite, delete, replace), tandis que le dictionnaire `summary` contient des statistiques détaillées sur chaque snapshot.

Pour comprendre comment Iceberg gère efficacement le voyage dans le temps, il est utile de visualiser la chaîne de snapshots. Chaque snapshot pointe vers un ensemble de manifests qui, à leur tour, référencent les fichiers de données. Lorsqu'une requête spécifie un snapshot historique, Iceberg navigue directement vers les fichiers pertinents sans avoir à parcourir l'historique complet.

Effectuons une modification pour créer un nouveau snapshot, puis voyageons dans le temps :

```
# Mise à jour : augmenter la valeur vie des clients Premium
spark.sql("""
    UPDATE nessie.ecommerce.clients
    SET valeur_vie = valeur_vie * 1.1
    WHERE segment = 'Premium'
```

```
""")

print("Mise à jour effectuée - nouveau snapshot créé")

# Consulter l'historique mis à jour
spark.sql("""
    SELECT snapshot_id, committed_at, operation
    FROM nessie.ecommerce.clients.snapshots
    ORDER BY committed_at DESC
""").show()
```

Pour consulter l'état de la table avant la mise à jour, utilisez la syntaxe de voyage dans le temps :

```
# Récupérer l'identifiant du premier snapshot
premier_snapshot = spark.sql("""
    SELECT snapshot_id
    FROM nessie.ecommerce.clients.snapshots
    ORDER BY committed_at ASC
    LIMIT 1
""").collect()[0][0]

# Requête sur l'état historique
spark.sql(f"""
    SELECT prenom, nom, segment, valeur_vie
    FROM nessie.ecommerce.clients
    VERSION AS OF {premier_snapshot}
    WHERE segment = 'Premium'
    ORDER BY valeur_vie DESC
""").show()

# Comparer avec l'état actuel
spark.sql("""
    SELECT prenom, nom, segment, valeur_vie
    FROM nessie.ecommerce.clients
    WHERE segment = 'Premium'
    ORDER BY valeur_vie DESC
""").show()
```

Démonstration de l'évolution de schéma

L'évolution de schéma (schema evolution) constitue une autre capacité distinctive d'Apache Iceberg. Ajoutons une nouvelle colonne à notre table sans perturber les données existantes :

```
# Ajouter une colonne pour le canal d'acquisition
spark.sql("""
    ALTER TABLE nessie.ecommerce.clients
    ADD COLUMN canal_acquisition STRING AFTER segment
""")

# Vérifier le schéma mis à jour
spark.sql("DESCRIBE nessie.ecommerce.clients").show()

# Mettre à jour les enregistrements existants
spark.sql("""
    UPDATE nessie.ecommerce.clients
```

```
    SET canal_acquisition = 'Migration historique'
    WHERE canal_acquisition IS NULL
""")

# Insérer de nouveaux enregistrements avec la nouvelle colonne
nouveaux_clients = [
    Row(1011, "Alexandre", "Roy", "alex.roy@courriel.ca", "QC",
        date(2024, 4, 15), "Premium", "Site web", 9500.00),
    Row(1012, "Priya", "Sharma", "priya.sharma@email.ca", "ON",
        date(2024, 4, 18), "Standard", "Référence", 3100.00),
]

schema_etendu = StructType([
    StructField("client_id", IntegerType(), False),
    StructField("prenom", StringType(), True),
    StructField("nom", StringType(), True),
    StructField("courriel", StringType(), True),
    StructField("province", StringType(), True),
    StructField("date_inscription", DateType(), True),
    StructField("segment", StringType(), True),
    StructField("canal_acquisition", StringType(), True),
    StructField("valeur_vie", DoubleType(), True),
])

df_nouveaux = spark.createDataFrame(nouveaux_clients, schema_etendu)
df_nouveaux.writeTo("nessie.ecommerce.clients").append()

# Valider l'insertion
spark.sql("""
    SELECT prenom, nom, canal_acquisition, valeur_vie
    FROM nessie.ecommerce.clients
    WHERE canal_acquisition != 'Migration historique'
""").show()
```

Étude de cas : Desjardins – Évolution de schéma sans interruption
Secteur : Services financiers
Défi : Ajouter des attributs de conformité réglementaire à des tables contenant des milliards d'enregistrements sans interrompre les opérations analytiques quotidiennes.
Solution : Migration vers Apache Iceberg pour bénéficier de l'évolution de schéma native. Les nouvelles colonnes sont ajoutées à chaud, et les processus de remplissage (backfill) s'exécutent en parallèle des requêtes de lecture.
Résultats : Zéro temps d'arrêt pendant les modifications de schéma, réduction de 80 % du temps de mise en production des nouveaux attributs de données.

Création d'une table de transactions

Pour enrichir notre démonstration, créons une table de transactions liée aux clients :

```
# Création de la table transactions
spark.sql("""
    CREATE TABLE IF NOT EXISTS nessie.ecommerce.transactions (
        transaction_id BIGINT,
        client_id BIGINT,
```

```
        date_transaction TIMESTAMP,
        montant DOUBLE,
        categorie STRING,
        mode_paiement STRING,
        statut STRING
    )
    USING iceberg
    PARTITIONED BY (days(date_transaction), categorie)
    TBLPROPERTIES (
        'write.format.default' = 'parquet',
        'write.parquet.compression-codec' = 'zstd'
    )
""")

# Générer des données de transactions
from random import randint, choice, uniform
from datetime import timedelta

transactions = []
categories = ['Électronique', 'Vêtements', 'Alimentation', 'Maison', 'Sports']
modes_paiement = ['Carte crédit', 'Carte débit', 'PayPal', 'Virement']
statuts = ['Complétée', 'Complétée', 'Complétée', 'Remboursée', 'En attente']

for i in range(1, 101):
    transactions.append(Row(
        transaction_id=10000 + i,
        client_id=1001 + (i % 12),
        date_transaction=datetime(2024, 1, 1) + timedelta(days=randint(0, 120)),
        montant=round(uniform(25.0, 500.0), 2),
        categorie=choice(categories),
        mode_paiement=choice(modes_paiement),
        statut=choice(statuts)
    ))

schema_transactions = StructType([
    StructField("transaction_id", IntegerType(), False),
    StructField("client_id", IntegerType(), True),
    StructField("date_transaction", TimestampType(), True),
    StructField("montant", DoubleType(), True),
    StructField("categorie", StringType(), True),
    StructField("mode_paiement", StringType(), True),
    StructField("statut", StringType(), True),
])

df_transactions = spark.createDataFrame(transactions, schema_transactions)
df_transactions.writeTo("nessie.ecommerce.transactions").append()

print(f"Insertion de {len(transactions)} transactions réussie")

# Requête analytique joignant les deux tables
spark.sql("""
    SELECT
        c.province,
        c.segment,
        COUNT(DISTINCT t.transaction_id) as nb_transactions,
        ROUND(SUM(t.montant), 2) as revenu_total,
        ROUND(AVG(t.montant), 2) as panier_moyen
    FROM nessie.ecommerce.clients c
    JOIN nessie.ecommerce.transactions t ON c.client_id = t.client_id
```

```
    WHERE t.statut = 'Complétée'
    GROUP BY c.province, c.segment
    ORDER BY revenu_total DESC
""").show()
```

Opérations avancées : MERGE INTO

L'instruction `MERGE INTO` constitue l'une des capacités les plus puissantes d'Iceberg pour les scénarios d'upsert (insertion ou mise à jour). Cette opération atomique permet de synchroniser une table cible avec une source de données en une seule transaction :

```
# Créer une table temporaire avec des mises à jour
spark.sql("""
    CREATE OR REPLACE TEMPORARY VIEW mises_a_jour_clients AS
    SELECT * FROM VALUES
        (1001, 'Marie', 'Tremblay-Roy', 'marie.tremblay@nouveau.ca', 'QC',
         DATE('2024-01-15'), 'Enterprise', 'Référence', 25000.00),
        (1013, 'Lucas', 'Bergeron', 'lucas.bergeron@courriel.ca', 'QC',
         DATE('2024-05-01'), 'Standard', 'Site web', 1500.00)
    AS t(client_id, prenom, nom, courriel, province, date_inscription,
         segment, canal_acquisition, valeur_vie)
""")

# Exécuter le MERGE INTO
spark.sql("""
    MERGE INTO nessie.ecommerce.clients AS cible
    USING mises_a_jour_clients AS source
    ON cible.client_id = source.client_id
    WHEN MATCHED THEN UPDATE SET
        nom = source.nom,
        courriel = source.courriel,
        segment = source.segment,
        valeur_vie = source.valeur_vie
    WHEN NOT MATCHED THEN INSERT *
""")

# Vérifier les résultats
spark.sql("""
    SELECT client_id, prenom, nom, segment, valeur_vie
    FROM nessie.ecommerce.clients
    WHERE client_id IN (1001, 1013)
""").show()
```

Cette opération illustre plusieurs comportements importants : - Le client 1001 existant a été mis à jour avec les nouvelles valeurs - Le client 1013, absent de la table, a été inséré - L'ensemble de l'opération s'est exécuté de manière atomique, créant un nouveau snapshot

Exploration des fichiers de métadonnées

Pour approfondir la compréhension du fonctionnement interne d'Iceberg, explorons les tables de métadonnées :

```
# Statistiques sur les fichiers de données
spark.sql("""
```

```
    SELECT
        file_path,
        file_format,
        record_count,
        file_size_in_bytes / 1024 as taille_ko,
        lower_bounds,
        upper_bounds
    FROM nessie.ecommerce.clients.files
""").show(truncate=False)

# Historique des partitions
spark.sql("""
    SELECT
        partition,
        record_count,
        file_count
    FROM nessie.ecommerce.clients.partitions
""").show(truncate=False)

# Manifests et leur contenu
spark.sql("""
    SELECT
        path,
        length,
        partition_spec_id,
        added_snapshot_id,
        added_data_files_count,
        existing_data_files_count,
        deleted_data_files_count
    FROM nessie.ecommerce.clients.manifests
""").show(truncate=False)
```

Ces tables de métadonnées exposent les détails internes de la structure Iceberg, permettant une observabilité complète sur l'état de vos tables.

Utilisation des branches Nessie (optionnel)

Le catalogue Nessie apporte une dimension supplémentaire : le versionnement des données à la manière de Git. Cette capacité permet de créer des branches pour expérimenter sans affecter les données de production :

```
# Créer une branche d'expérimentation
spark.sql("CREATE BRANCH IF NOT EXISTS experimentation IN nessie")

# Basculer vers la branche
spark.sql("USE REFERENCE experimentation IN nessie")

# Effectuer des modifications sur la branche
spark.sql("""
    DELETE FROM nessie.ecommerce.clients
    WHERE segment = 'Standard'
""")

# Vérifier l'état de la branche
spark.sql("""
    SELECT segment, COUNT(*) as nombre
```

```
    FROM nessie.ecommerce.clients
    GROUP BY segment
""").show()

# Revenir à la branche principale (main)
spark.sql("USE REFERENCE main IN nessie")

# Vérifier que les données de production sont intactes
spark.sql("""
    SELECT segment, COUNT(*) as nombre
    FROM nessie.ecommerce.clients
    GROUP BY segment
""").show()
```

Cette fonctionnalité s'avère particulièrement précieuse pour : - Les environnements de développement et de test isolés - La validation des transformations avant déploiement en production - Les analyses exploratoires nécessitant des modifications temporaires

---

IV.3.3 Lecture des tables Iceberg dans Dremio

Dremio représente la couche de fédération et d'accélération de requêtes dans notre architecture Lakehouse. Cette plateforme permet d'unifier l'accès aux données provenant de sources multiples, d'optimiser les performances grâce aux réflexions de données et de fournir une interface SQL accessible aux analystes métier. L'intégration native avec Apache Iceberg fait de Dremio un choix privilégié pour la consommation des données du Lakehouse.

L'architecture de Dremio repose sur plusieurs composants clés qui contribuent à ses performances remarquables :

- **Moteur de requêtes distribué** : Basé sur Apache Arrow, le moteur de Dremio traite les données en mémoire avec un format colonnaire optimisé pour les processeurs modernes (vectorisation SIMD).
- **Couche de planification intelligente** : Analyse les requêtes pour déterminer le plan d'exécution optimal, incluant la substitution automatique par des réflexions précalculées.
- **Gestion des réflexions** : Système de matérialisations automatiques qui crée des vues physiques optimisées sous forme de tables Iceberg.
- **Connecteurs natifs** : Support direct de nombreuses sources de données, évitant les couches d'abstraction coûteuses.

Pour les utilisateurs familiers avec les entrepôts de données traditionnels, Dremio offre une expérience SQL complète tout en éliminant la nécessité de déplacer les données vers une infrastructure propriétaire. Les données restent sur votre Lakehouse (stockage objet), et Dremio fournit l'intelligence de requête au-dessus.

Configuration initiale de Dremio

Accédez à l'interface Dremio via `http://localhost:9047`. Lors de votre première connexion, vous serez invité à créer un compte administrateur. Complétez l'assistant de configuration avec les informations suivantes :

1. **Création du compte** : Définissez un nom d'utilisateur (par exemple, `admin`) et un mot de passe robuste.
2. **Configuration du projet** : Acceptez les paramètres par défaut pour l'environnement de développement.

Connexion au catalogue Nessie

Dremio prend en charge nativement les catalogues Nessie, offrant une intégration transparente avec nos tables Iceberg. Pour configurer cette connexion :

1. Cliquez sur **Add Source** dans le panneau de navigation gauche.
2. Sélectionnez **Nessie** dans la liste des sources disponibles.
3. Configurez les paramètres suivants :

Paramètre	Valeur
Name	`lakehouse`
Nessie Endpoint URL	`http://nessie:19120/api/v2`
Authentication	None
AWS Root Path	`warehouse`
AWS Access Key	`admin`
AWS Secret Key	`password123`
Connection Properties	`fs.s3a.endpoint=http://minio:9000`
	`fs.s3a.path.style.access=true`
Encrypt connection	Désactivé

4. Cliquez sur **Save** pour établir la connexion.

Après la sauvegarde, Dremio découvre automatiquement les espaces de noms et les tables présents dans le catalogue Nessie. Vous devriez voir apparaître l'espace de noms `ecommerce` avec les tables `clients` et `transactions`.

Exploration des données via l'interface SQL

Dremio offre un éditeur SQL intégré permettant d'interroger directement les tables Iceberg. Accédez à l'onglet **SQL Runner** et exécutez les requêtes suivantes :

```
-- Exploration de la table clients
SELECT * FROM lakehouse.ecommerce.clients
LIMIT 10;

-- Statistiques par province
SELECT
    province,
    COUNT(*) AS nombre_clients,
    ROUND(AVG(valeur_vie), 2) AS valeur_moyenne,
```

```
    ROUND(SUM(valeur_vie), 2) AS valeur_totale
FROM lakehouse.ecommerce.clients
GROUP BY province
ORDER BY valeur_totale DESC;
```

Fonctionnalités Iceberg dans Dremio

Dremio expose les fonctionnalités avancées d'Iceberg à travers des extensions SQL propriétaires et des tables de métadonnées.

Consultation de l'historique des snapshots :

```
-- Historique des modifications
SELECT * FROM TABLE(
    table_history('lakehouse.ecommerce.clients')
);

-- Liste des snapshots
SELECT * FROM TABLE(
    table_snapshot('lakehouse.ecommerce.clients')
);
```

Voyage dans le temps :

```
-- Requête sur un snapshot spécifique
SELECT * FROM lakehouse.ecommerce.clients
AT SNAPSHOT '1234567890123456789';

-- Requête à un moment précis
SELECT * FROM lakehouse.ecommerce.clients
AT BRANCH main
AS OF '2024-04-01 00:00:00';
```

Exploration des métadonnées de fichiers :

```
-- Statistiques sur les fichiers de données
SELECT
    file_path,
    file_format,
    record_count,
    file_size_in_bytes
FROM TABLE(
    table_files('lakehouse.ecommerce.clients')
);
```

Création de vues virtuelles

Les vues virtuelles (Virtual Datasets) de Dremio permettent de créer des abstractions sémantiques au-dessus des tables Iceberg, simplifiant l'accès pour les utilisateurs métier :

```
-- Vue consolidée clients-transactions
CREATE VDS lakehouse.analytique.resume_clients AS
```

```
SELECT
    c.client_id,
    c.prenom || ' ' || c.nom AS nom_complet,
    c.province,
    c.segment,
    c.valeur_vie,
    COUNT(t.transaction_id) AS nombre_transactions,
    COALESCE(SUM(t.montant), 0) AS montant_total_achats,
    COALESCE(AVG(t.montant), 0) AS panier_moyen,
    MAX(t.date_transaction) AS derniere_transaction
FROM lakehouse.ecommerce.clients c
LEFT JOIN lakehouse.ecommerce.transactions t
    ON c.client_id = t.client_id
    AND t.statut = 'Complétée'
GROUP BY
    c.client_id, c.prenom, c.nom,
    c.province, c.segment, c.valeur_vie;
```

Configuration des réflexions de données

Les réflexions de données (Data Reflections) constituent la fonctionnalité d'accélération signature de Dremio. Elles créent des matérialisations optimisées des données sous forme de tables Iceberg, que Dremio substitue automatiquement lors de l'exécution des requêtes.

Le principe de fonctionnement des réflexions est élégant : lorsque vous activez une réflexion sur une vue ou une table, Dremio matérialise les données dans un format optimisé. Lors de l'exécution ultérieure de requêtes, le planificateur évalue si la réflexion peut satisfaire la requête. Si c'est le cas, la requête est automatiquement redirigée vers la réflexion, offrant des temps de réponse considérablement réduits.

Dremio propose deux types de réflexions :

Réflexions brutes (Raw Reflections) : Matérialisent l'ensemble des données d'une table ou vue. Elles accélèrent les requêtes qui lisent un grand nombre de colonnes ou qui nécessitent un scan complet. Les réflexions brutes sont particulièrement utiles pour : - Les requêtes ad hoc imprévisibles - Les exports de données volumineuses - Les jointures nécessitant l'accès à toutes les colonnes

Réflexions d'agrégation (Aggregation Reflections) : Précalculent des agrégations selon des dimensions définies. Elles accélèrent dramatiquement les requêtes analytiques typiques (sommes, moyennes, comptages). Les réflexions d'agrégation excellent pour : - Les tableaux de bord avec des KPI prédéfinis - Les rapports périodiques standardisés - Les requêtes avec des GROUP BY fréquents

Pour activer les réflexions sur notre vue :

1. Naviguez vers la vue `resume_clients` dans l'explorateur de données.
2. Cliquez sur l'onglet **Reflections**.
3. Activez **Raw Reflections** pour matérialiser l'ensemble des données.
4. Configurez **Aggregation Reflections** avec les dimensions et mesures suivantes :

Type	Colonnes
Dimensions	`province`, `segment`
Mesures	`SUM(montant_total_achats)`, `COUNT(nombre_transactions)`, `AVG(panier_moyen)`

5. Définissez une politique de rafraîchissement (par exemple, toutes les heures).

La configuration du rafraîchissement mérite une attention particulière. Dremio supporte plusieurs modes : - **Rafraîchissement complet** : Reconstruit entièrement la réflexion à chaque cycle - **Rafraîchissement incrémental** : Met à jour uniquement les partitions modifiées (recommandé pour Iceberg) - **Rafraîchissement basé sur les snapshots** : Détecte automatiquement les nouveaux snapshots Iceberg

> **Performance**
> Les réflexions de données peuvent améliorer les performances des requêtes de 10 à 100 fois selon la complexité des agrégations et le volume de données. Dans notre environnement de démonstration, l'impact sera modeste, mais sur des tables de plusieurs téraoctets, les gains sont substantiels. Une étude de Dremio (2024) rapporte des temps de réponse passant de plusieurs minutes à moins d'une seconde pour des requêtes d'agrégation sur des tables de 500 Go.

Connexion à des sources externes

Dremio excelle dans la fédération de données provenant de sources hétérogènes. Vous pouvez ajouter des connexions vers :

- **Bases de données relationnelles** : PostgreSQL, MySQL, SQL Server, Oracle
- **Entrepôts de données infonuagiques** : Snowflake, BigQuery, Redshift
- **Stockage objet** : Amazon S3, Azure Blob Storage, Google Cloud Storage
- **Systèmes de fichiers** : HDFS, NAS, partages réseau

Cette capacité permet de créer des requêtes fédérées joignant des tables Iceberg du Lakehouse avec des données opérationnelles sans déplacer physiquement les données.

Requêtes fédérées et virtualisation

L'un des avantages majeurs de Dremio réside dans sa capacité à exécuter des requêtes fédérées transparentes. Imaginons que vous ajoutiez une source PostgreSQL contenant des données de référence :

```
-- Requête fédérée combinant Iceberg et PostgreSQL
SELECT
    c.client_id,
    c.nom,
    c.province,
    ref.nom_region,
    ref.fuseau_horaire,
    SUM(t.montant) AS total_achats
FROM lakehouse.ecommerce.clients c
JOIN postgres_source.public.regions ref
    ON c.province = ref.code_province
LEFT JOIN lakehouse.ecommerce.transactions t
    ON c.client_id = t.client_id
```

```
GROUP BY c.client_id, c.nom, c.province,
         ref.nom_region, ref.fuseau_horaire
ORDER BY total_achats DESC;
```

Dremio optimise automatiquement l'exécution en : - Poussant les filtres vers les sources (predicate push-down) - Parallélisant l'accès aux différentes sources - Utilisant les réflexions disponibles pour accélérer les agrégations

Gestion des droits d'accès

Dremio intègre un système de contrôle d'accès granulaire permettant de sécuriser l'accès aux données :

```
-- Accorder l'accès en lecture sur un espace de noms
GRANT SELECT ON lakehouse.ecommerce TO analystes;

-- Créer une politique de masquage des données sensibles
ALTER TABLE lakehouse.ecommerce.clients
SET MASKING POLICY courriel_masque ON courriel;

-- Limiter l'accès par lignes (Row-Level Security)
ALTER TABLE lakehouse.ecommerce.clients
SET ROW ACCESS POLICY filtre_province
ON province USING (province = CURRENT_USER_PROVINCE());
```

Ces fonctionnalités de gouvernance s'avèrent essentielles pour les déploiements en entreprise où la conformité réglementaire (Loi 25, RGPD) impose des contrôles stricts sur l'accès aux données personnelles.

Optimisation des requêtes avec EXPLAIN

Dremio fournit des outils de diagnostic permettant de comprendre et d'optimiser l'exécution des requêtes :

```
-- Analyser le plan d'exécution
EXPLAIN PLAN FOR
SELECT province, segment, COUNT(*), AVG(valeur_vie)
FROM lakehouse.ecommerce.clients
GROUP BY province, segment;

-- Voir le plan avec les coûts estimés
EXPLAIN PLAN INCLUDING ALL ATTRIBUTES FOR
SELECT c.province, SUM(t.montant)
FROM lakehouse.ecommerce.clients c
JOIN lakehouse.ecommerce.transactions t
    ON c.client_id = t.client_id
GROUP BY c.province;
```

L'analyse du plan d'exécution révèle : - Les opérations de lecture (scans) sur les tables Iceberg - L'utilisation des statistiques de colonnes pour le filtrage - La substitution par des réflexions lorsque disponibles - Les opérations de brassage (shuffle) entre les nœuds

Profils de requête et métriques

Après l'exécution d'une requête, Dremio capture un profil détaillé accessible via l'interface utilisateur (Jobs → Sélectionner la requête → View Profile). Ce profil contient :

- **Temps par phase** : Planification, exécution, écriture des résultats
- **Métriques d'entrées/sorties** : Octets lus/écrits, enregistrements traités
- **Utilisation des réflexions** : Indique si une réflexion a été utilisée
- **Parallélisme** : Nombre de threads et de fragments d'exécution

Ces informations permettent d'identifier les goulots d'étranglement et d'optimiser les requêtes ou les configurations de réflexions.

IV.3.4 Création d'un tableau de bord BI

La valeur ultime d'un Lakehouse réside dans sa capacité à alimenter des analyses métier exploitables. Dans cette section, nous connecterons Dremio à des outils d'intelligence d'affaires pour créer des visualisations interactives. Nous explorerons deux approches : Apache Superset (solution à code source ouvert) et les options de connexion aux outils commerciaux comme Power BI et Tableau.

Option 1 : Apache Superset

Apache Superset est une plateforme de visualisation de données moderne, à code source ouvert, capable de créer des tableaux de bord interactifs et des explorations de données ad hoc. Développé initialement par Airbnb et maintenant projet de la Fondation Apache, Superset s'est imposé comme l'alternative open source de référence aux solutions commerciales de BI.

Les points forts de Superset pour un contexte Lakehouse incluent :

- **Interface de création de graphiques intuitive** : Les utilisateurs non techniques peuvent créer des visualisations sans écrire de SQL
- **Support SQL complet** : Les analystes avancés bénéficient d'un éditeur SQL intégré avec autocomplétion
- **Tableaux de bord interactifs** : Filtres croisés, drill-down et paramètres dynamiques
- **Sécurité robuste** : Contrôle d'accès basé sur les rôles, intégration LDAP/OAuth
- **Extensibilité** : Architecture de plugins permettant d'ajouter de nouveaux types de visualisations

Ajout de Superset à l'environnement Docker

Étendez votre fichier `docker-compose.yml` avec le service Superset :

```
  superset:
    image: apache/superset:latest
    container_name: superset
    depends_on:
      - dremio
    environment:
      SUPERSET_SECRET_KEY: 'votre_cle_secrete_complexe_ici'
      SUPERSET_LOAD_EXAMPLES: 'no'
    ports:
      - "8088:8088"
    volumes:
```

```
      - superset-data:/app/superset_home
    networks:
      - lakehouse-network
    command: >
      bash -c "
        superset db upgrade &&
        superset fab create-admin --username admin --firstname Admin --lastname User --
email admin@exemple.ca --password admin123 &&
        superset init &&
        superset run -h 0.0.0.0 -p 8088
      "

volumes:
  # ... volumes existants ...
  superset-data:
```

Cette configuration effectue automatiquement l'initialisation de la base de données Superset, crée un compte administrateur et démarre le serveur web. En production, vous sépareriez ces étapes et utiliseriez des variables d'environnement sécurisées pour les identifiants.

Redémarrez l'environnement pour inclure Superset :

```
docker-compose up -d superset
```

L'initialisation de Superset prend généralement 2 à 3 minutes. Vous pouvez suivre la progression avec `docker logs -f superset`.

Configuration de la connexion Dremio dans Superset

1. Accédez à Superset via `http://localhost:8088` et connectez-vous (`admin` / `admin123`).
2. Naviguez vers **Settings → Database Connections → + Database**.
3. Sélectionnez **Other** comme type de base de données.
4. Utilisez la chaîne de connexion SQLAlchemy suivante :

```
dremio+flight://admin:votre_mot_de_passe@dremio:32010/dremio?UseEncryption=false
```

> **Note** : Le pilote Dremio Flight pour SQLAlchemy (`sqlalchemy-dremio`) doit être installé dans le conteneur Superset. Pour une installation persistante, créez une image Docker personnalisée basée sur `apache/superset` avec le pilote préinstallé :
>
> ```
> FROM apache/superset:latest
> USER root
> RUN pip install sqlalchemy-dremio
> USER superset
> ```

5. Testez la connexion et enregistrez la configuration.

Configuration avancée : Modèle de données Superset

Superset utilise le concept de « datasets » (jeux de données) pour exposer les tables aux créateurs de tableaux de bord. Pour optimiser l'expérience utilisateur :

1. Créez un dataset pointant vers la vue `resume_clients` de Dremio.
2. Configurez les colonnes avec des métadonnées enrichies :
   - Définissez des libellés conviviaux (« Province » au lieu de « province »)
   - Marquez les colonnes numériques comme métriques
   - Définissez les formats d'affichage (devise, pourcentage)
3. Créez des métriques calculées récurrentes directement dans le dataset :
   - Revenu moyen par client : `SUM(montant_total_achats) / COUNT(client_id)`
   - Taux de conversion : `COUNT(CASE WHEN nombre_transactions > 0 THEN 1 END) / COUNT(*)`

Création d'un tableau de bord analytique

Avec la connexion établie, créez un tableau de bord présentant les indicateurs clés de performance (KPI) de notre plateforme de commerce électronique :

Graphique 1 — Répartition des clients par segment

1. Cliquez sur **+ Chart** et sélectionnez la source de données `resume_clients`.
2. Choisissez le type **Pie Chart**.
3. Configurez :
   - **Dimension** : `segment`
   - **Metric** : `COUNT(*)`
4. Enregistrez et ajoutez au tableau de bord.

Graphique 2 — Revenus par province

1. Créez un nouveau graphique de type **Bar Chart**.
2. Configurez :
   - **X-Axis** : `province`
   - **Metric** : `SUM(montant_total_achats)`
   - **Sort** : Décroissant par métrique
3. Ajoutez au tableau de bord.

Graphique 3 — Évolution temporelle des transactions

Pour ce graphique, nous avons besoin d'une requête personnalisée :

```
SELECT
    DATE_TRUNC('week', date_transaction) AS semaine,
    categorie,
    SUM(montant) AS revenus,
    COUNT(*) AS volume
FROM lakehouse.ecommerce.transactions
WHERE statut = 'Complétée'
GROUP BY DATE_TRUNC('week', date_transaction), categorie
ORDER BY semaine;
```

1. Créez un graphique de type **Line Chart**.
2. Utilisez la requête SQL personnalisée.
3. Configurez :
   - **X-Axis** : `semaine`
   - **Y-Axis** : `revenus`

- **Group By** : `categorie`

4. Ajoutez au tableau de bord.

Option 2 : Connexion Power BI

Microsoft Power BI représente l'outil d'intelligence d'affaires le plus répandu dans les entreprises, particulièrement au Canada où l'écosystème Microsoft bénéficie d'une adoption massive. L'intégration avec Dremio offre aux organisations utilisant Power BI un chemin naturel vers le Lakehouse sans imposer de changement d'outils aux utilisateurs métier.

Prérequis : - Power BI Desktop installé (version Windows) - Pilote ODBC Dremio téléchargé depuis le site officiel Dremio - Compte Dremio avec les permissions appropriées

L'architecture de connexion entre Power BI et Dremio s'appuie sur le protocole ODBC (Open Database Connectivity), un standard de l'industrie pour la connectivité aux bases de données. Cette approche garantit une compatibilité large et une maintenance simplifiée.

Installation du pilote ODBC Dremio

1. Téléchargez le pilote ODBC Dremio depuis https://www.dremio.com/drivers/
2. Exécutez l'installateur en tant qu'administrateur
3. Configurez un DSN (Data Source Name) système :
   - Ouvrez ODBC Data Source Administrator (64-bit)
   - Cliquez sur « Add » dans l'onglet « System DSN »
   - Sélectionnez « Dremio ODBC Driver »
   - Configurez les paramètres de connexion

Configuration de la connexion :

1. Dans Power BI Desktop, sélectionnez **Get Data → More → ODBC**.
2. Créez une nouvelle source de données ODBC avec les paramètres :
   - **DSN** : Créez un DSN système pointant vers `localhost:31010`
   - **Authentication** : Basic avec vos identifiants Dremio
3. Naviguez dans le catalogue Dremio et sélectionnez les tables ou vues à importer.
4. Choisissez le mode de connectivité (Import ou DirectQuery).

Mode DirectQuery vs Import

Le choix entre DirectQuery et Import constitue une décision architecturale importante qui influence les performances, la fraîcheur des données et l'expérience utilisateur :

Aspect	DirectQuery	Import
Fraîcheur des données	Temps réel	Selon planification
Performance des requêtes	Variable (dépend de la source)	Optimale (données en mémoire)
Fonctionnalités DAX	Limitées	Complètes
Taille des données	Illimitée	Limitée par la mémoire
Coût de licence	Standard	Standard
Charge sur la source	Chaque interaction	Uniquement lors du rafraîchissement

Pour un Lakehouse Iceberg avec Dremio, le mode **DirectQuery** est généralement recommandé car les réflexions Dremio compensent les latences de requête, et vous bénéficiez des données actualisées sans planification d'actualisation complexe. Cependant, pour des rapports nécessitant des calculs DAX complexes ou des visualisations avec de nombreuses interactions utilisateur, le mode Import peut offrir une meilleure expérience.

Optimisation pour Power BI

Plusieurs pratiques optimisent l'intégration Power BI-Dremio :

- **Créez des vues optimisées** : Préparez dans Dremio des vues correspondant exactement aux besoins des rapports, évitant les transformations côté Power BI
- **Utilisez les réflexions d'agrégation** : Configurez des réflexions alignées avec les agrégations fréquentes dans vos rapports
- **Limitez les colonnes** : Sélectionnez uniquement les colonnes nécessaires pour réduire le transfert de données
- **Évitez les filtres complexes** : Les prédicats simples sont plus efficacement poussés vers Dremio

> **Étude de cas : Banque Nationale – Migration vers Power BI Direct Query**
> *Secteur* : Services financiers
> *Défi* : Migrer des centaines de rapports SSRS vers Power BI tout en modernisant l'infrastructure de données sous-jacente vers un Lakehouse.
> *Solution* : Déploiement progressif de Dremio comme couche sémantique unifiée, permettant aux rapports Power BI d'accéder aux données Iceberg via DirectQuery sans réécrire les modèles de données existants.
> *Résultats* : Migration de 340 rapports en 6 mois avec une amélioration moyenne de 40 % des temps de réponse grâce aux réflexions Dremio.

Option 3 : Connexion Tableau

Tableau représente la référence en matière de visualisation de données analytiques, particulièrement apprécié pour ses capacités d'exploration visuelle et son approche « glisser-déposer » intuitive. L'intégration avec Dremio combine la puissance d'exploration de Tableau avec les capacités d'accélération et de fédération du Lakehouse.

Configuration de la connexion :

1. Dans Tableau Desktop, sélectionnez **Connect → To a Server → More → Dremio**.

2. Entrez les paramètres de connexion :
   - **Server** : `localhost`
   - **Port** : `31010`
   - **Authentication** : Username and Password
3. Naviguez dans le catalogue et glissez les tables sur le canevas de modélisation.

Modélisation des données dans Tableau

Tableau offre plusieurs approches pour modéliser les relations entre tables :

- **Relations** : Approche recommandée depuis Tableau 2020.2, permettant des jointures dynamiques selon le contexte de la visualisation
- **Jointures physiques** : Jointures explicites définies au niveau de la source de données
- **Extraits** : Matérialisation locale des données pour des performances optimales hors ligne

Pour un Lakehouse Dremio, l'utilisation des **relations** avec une connexion **live** (DirectQuery équivalent) offre le meilleur équilibre entre fraîcheur des données et flexibilité de modélisation. Les réflexions Dremio compensent l'absence d'extraits locaux.

Optimisation des performances Tableau

Plusieurs techniques améliorent les performances de Tableau connecté à Dremio :

1. **Agrégations initiales** : Configurez Tableau pour utiliser les agrégations initiales (Initial SQL) afin d'exécuter des commandes préparatoires
2. **Context filters** : Utilisez les filtres de contexte pour réduire le volume de données avant les autres calculs
3. **Extraits hybrides** : Pour les données historiques stables, créez des extraits, tout en maintenant les données récentes en mode live
4. **Parallélisme** : Activez le parallélisme de requêtes dans les paramètres de source de données

> **Étude de cas : Air Canada – Tableaux de bord opérationnels**
> *Secteur* : Transport aérien
> *Défi* : Unifier les données de réservation, d'opérations et de fidélité provenant de dizaines de systèmes sources pour alimenter des tableaux de bord en temps quasi réel.
> *Solution* : Déploiement d'un Lakehouse Iceberg avec Dremio comme couche de fédération, connecté à Tableau pour la visualisation. Les réflexions Dremio préagrègent les métriques clés.
> *Résultats* : Temps de rafraîchissement des tableaux de bord réduit de 45 minutes à moins de 30 secondes. Consolidation de 23 sources de données en une interface analytique unifiée.

Considérations pour la production

Lors du passage à un environnement de production, plusieurs aspects méritent attention :

Sécurité des connexions

- Activez le chiffrement TLS pour toutes les connexions entre les outils BI et Dremio.
- Implémentez l'authentification unique (SSO) via SAML ou OpenID Connect.
- Configurez le contrôle d'accès basé sur les rôles (RBAC) dans Dremio pour limiter l'accès aux données sensibles.

Optimisation des performances

- Dimensionnez les réflexions Dremio en fonction des patrons de requêtes observés.
- Configurez des politiques de rafraîchissement adaptées à la fraîcheur requise des données.
- Utilisez le partitionnement Iceberg aligné avec les filtres fréquents des tableaux de bord.

Gouvernance des données

- Documentez les définitions des métriques dans le catalogue Dremio.
- Établissez des processus de certification des jeux de données.
- Implémentez le lignage des données pour tracer l'origine des métriques.

Bonnes pratiques pour les tableaux de bord Lakehouse

L'expérience accumulée dans les déploiements de Lakehouse révèle plusieurs bonnes pratiques pour maximiser la valeur des tableaux de bord analytiques :

1. Conception centrée sur les cas d'utilisation

Plutôt que de créer des tableaux de bord génériques, concentrez-vous sur des questions métier spécifiques. Un tableau de bord efficace répond à des questions précises : - Quels segments de clients génèrent le plus de revenus par province ? - Quelle est la tendance des paniers moyens cette semaine comparée au mois précédent ? - Quelles catégories de produits connaissent une croissance des ventes ?

2. Couche sémantique unifiée

Créez des vues virtuelles dans Dremio qui encapsulent la logique métier complexe. Ces vues servent de « contrat » entre les ingénieurs de données et les analystes :

```
-- Vue sémantique pour l'analyse des ventes
CREATE VDS lakehouse.semantique.fait_ventes AS
SELECT
    t.transaction_id,
    t.date_transaction,
    DATE_TRUNC('month', t.date_transaction) AS mois,
    DATE_TRUNC('week', t.date_transaction) AS semaine,
    t.montant,
    t.categorie,
    -- Dimensions enrichies
    c.segment AS segment_client,
    c.province,
    c.valeur_vie AS valeur_vie_client,
    -- Métriques dérivées
    CASE
        WHEN t.montant < 50 THEN 'Petit panier'
        WHEN t.montant < 150 THEN 'Panier moyen'
        ELSE 'Grand panier'
    END AS tranche_panier
FROM lakehouse.ecommerce.transactions t
JOIN lakehouse.ecommerce.clients c ON t.client_id = c.client_id
WHERE t.statut = 'Complétée';
```

3. Stratégie de rafraîchissement différenciée

Tous les tableaux de bord n'ont pas les mêmes exigences de fraîcheur :

Type de tableau de bord	Fraîcheur requise	Stratégie
Opérationnel (ventes du jour)	Quasi temps réel	DirectQuery + Réflexion horaire
Tactique (performance hebdomadaire)	Quotidienne	Import nocturne
Stratégique (tendances trimestrielles)	Hebdomadaire	Import planifié

4. Gestion des agrégations pré-calculées

Pour les tableaux de bord à forte charge, créez des tables agrégées dans Iceberg :

```
# Créer une table d'agrégation quotidienne
spark.sql("""
    CREATE TABLE IF NOT EXISTS nessie.ecommerce.agg_ventes_quotidiennes
    USING iceberg
    AS
    SELECT
        DATE(date_transaction) AS date_vente,
        categorie,
        c.province,
        c.segment,
        COUNT(*) AS nombre_transactions,
        SUM(montant) AS montant_total,
        AVG(montant) AS panier_moyen
    FROM nessie.ecommerce.transactions t
    JOIN nessie.ecommerce.clients c ON t.client_id = c.client_id
    WHERE t.statut = 'Complétée'
    GROUP BY DATE(date_transaction), categorie, c.province, c.segment
""")
```

Cette table peut ensuite être rafraîchie incrémentalement, offrant des performances de requête optimales pour les tableaux de bord.

5. Monitoring et alertes

Implémentez un suivi des métriques clés pour détecter les anomalies :

```
-- Requête de monitoring des ventes quotidiennes
SELECT
    DATE(date_transaction) AS date_vente,
    COUNT(*) AS nb_transactions,
    SUM(montant) AS total_ventes,
    -- Comparaison avec la moyenne mobile sur 7 jours
    AVG(SUM(montant)) OVER (
        ORDER BY DATE(date_transaction)
        ROWS BETWEEN 7 PRECEDING AND 1 PRECEDING
    ) AS moyenne_7j,
    -- Écart en pourcentage
    (SUM(montant) - AVG(SUM(montant)) OVER (
        ORDER BY DATE(date_transaction)
        ROWS BETWEEN 7 PRECEDING AND 1 PRECEDING
    )) / NULLIF(AVG(SUM(montant)) OVER (
        ORDER BY DATE(date_transaction)
        ROWS BETWEEN 7 PRECEDING AND 1 PRECEDING
    ), 0) * 100 AS ecart_pct
```

```
FROM lakehouse.ecommerce.transactions
WHERE statut = 'Complétée'
GROUP BY DATE(date_transaction)
ORDER BY date_vente DESC
LIMIT 30;
```

Étude de cas : Couche-Tard – Analyse multi-magasins
Secteur : Commerce de détail
Défi : Consolider les données de milliers de points de vente à travers l'Amérique du Nord pour des analyses de performance en temps quasi réel.
Solution : Architecture Lakehouse avec Iceberg pour le stockage historique, intégration Kafka pour l'ingestion en continu (détaillée au Volume III), et Dremio pour la couche sémantique unifiée alimentant des tableaux de bord Power BI.
Résultats : Réduction du délai de disponibilité des données de T+1 jour à moins de 15 minutes. Capacité d'analyse ad hoc sur 5 ans d'historique transactionnel sans impact sur les systèmes opérationnels.

IV.3.5 Résumé

Ce chapitre a concrétisé les concepts théoriques des chapitres précédents à travers une implémentation pratique complète. Nous avons parcouru l'ensemble du cycle de vie des données dans un Data Lakehouse moderne, de la configuration de l'infrastructure jusqu'à la création de visualisations analytiques.

Compétences acquises

À l'issue de ce chapitre, vous maîtrisez :

Configuration d'environnement - Déploiement d'un écosystème Lakehouse conteneurisé avec Docker Compose - Configuration de MinIO comme stockage objet compatible S3 - Intégration du catalogue Nessie avec versionnement des données - Orchestration des services Spark, Dremio et outils de visualisation

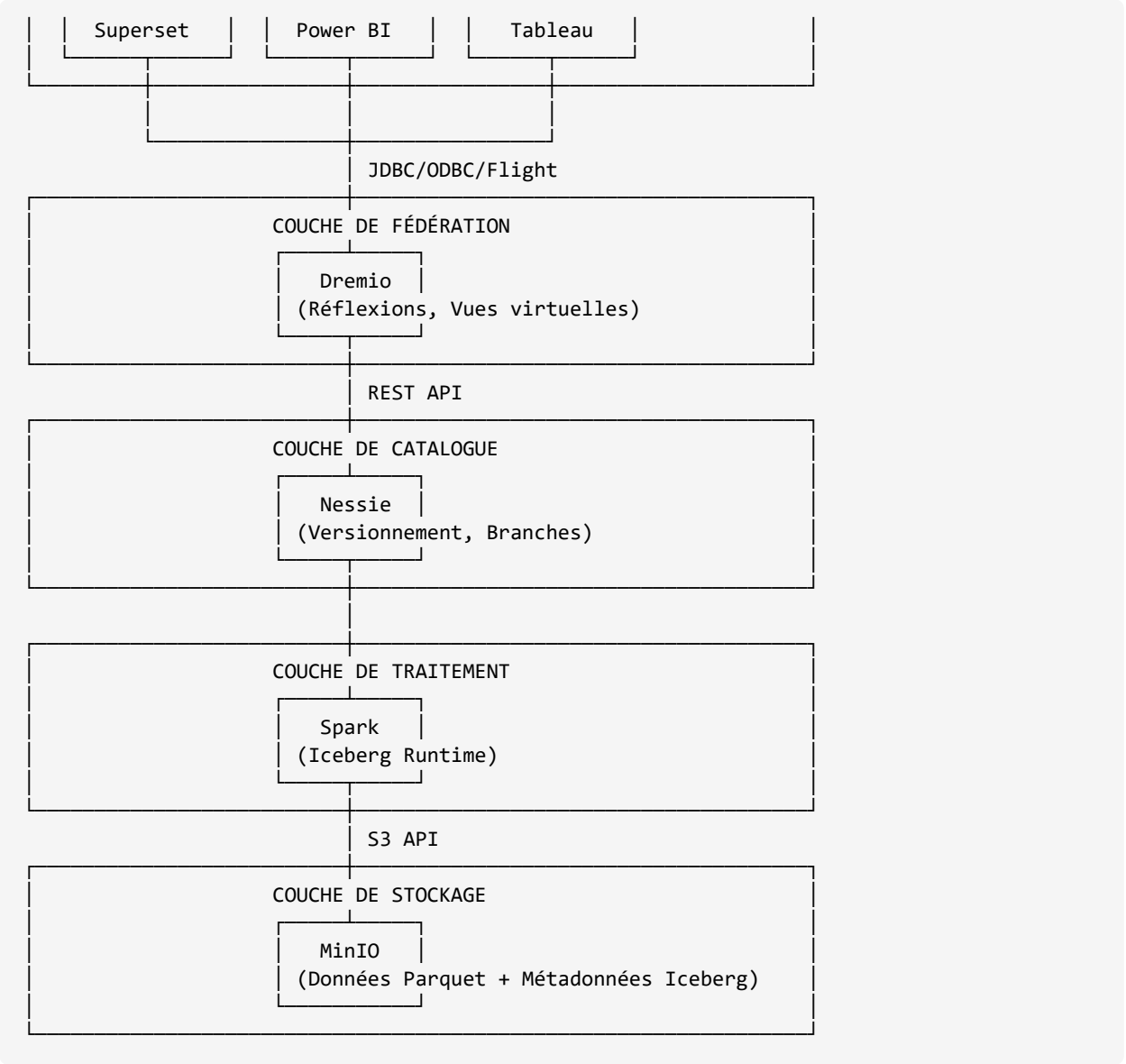
Manipulation de tables Iceberg avec Spark - Création d'espaces de noms et de tables avec partitionnement masqué - Insertion et mise à jour de données via les API DataFrame et SQL - Exploration des snapshots et utilisation du voyage dans le temps - Évolution de schéma sans interruption de service

Fédération et accélération avec Dremio - Configuration des connexions aux catalogues Iceberg - Création de vues virtuelles pour l'abstraction sémantique - Mise en place des réflexions de données pour l'accélération - Interrogation des métadonnées Iceberg via SQL étendu

Visualisation et intelligence d'affaires - Connexion d'Apache Superset, Power BI et Tableau à Dremio - Création de tableaux de bord interactifs - Choix entre les modes DirectQuery et Import - Considérations de sécurité et de gouvernance

Architecture mise en œuvre

```
┌─────────────────────────────────────────────────────────────────┐
│                    COUCHE DE CONSOMMATION                       │
│  ┌──────────────┐  ┌──────────────┐  ┌──────────────┐           │
```



Points clés à retenir

1. **L'écosystème Lakehouse est modulaire** : Chaque composant (stockage, catalogue, traitement, fédération, consommation) peut être remplacé indépendamment grâce aux standards ouverts comme la spécification Apache Iceberg.
2. **Le partitionnement masqué simplifie l'expérience utilisateur** : Les requêtes filtrent sur les colonnes métier (`date_inscription`) sans exposer la structure technique de partitionnement.
3. **Le voyage dans le temps offre une traçabilité complète** : Chaque modification crée un snapshot, permettant l'audit, la récupération et la comparaison des états historiques.
4. **L'évolution de schéma est une opération métadonnées** : Ajouter, renommer ou modifier des colonnes ne requiert pas de réécriture des données existantes.
5. **Dremio accélère sans dupliquer** : Les réflexions de données créent des matérialisations intelligentes sur le Lakehouse lui-même, évitant la prolifération des copies de données.

6. **La fédération de données élimine les silos** : Dremio permet de joindre des données Iceberg avec d'autres sources sans déplacement physique, simplifiant l'architecture globale.
7. **Le versionnement Nessie apporte les pratiques DevOps aux données** : Les branches permettent d'expérimenter sans risque et de valider les transformations avant déploiement.
8. **Les outils BI s'intègrent nativement** : Power BI, Tableau et Superset se connectent à Dremio via des protocoles standards (ODBC, JDBC, Flight), bénéficiant des optimisations du Lakehouse.

Commandes et syntaxes essentielles

Voici un aide-mémoire des commandes clés utilisées dans ce chapitre :

Gestion des espaces de noms et tables (Spark SQL)

```
-- Espaces de noms
CREATE NAMESPACE nessie.mon_namespace;
SHOW NAMESPACES IN nessie;
DROP NAMESPACE nessie.mon_namespace;

-- Tables
CREATE TABLE nessie.ns.table (...) USING iceberg PARTITIONED BY (...);
DESCRIBE EXTENDED nessie.ns.table;
ALTER TABLE nessie.ns.table ADD COLUMN nouvelle_col STRING;
DROP TABLE nessie.ns.table;
```

Opérations de données

```
-- Insertions
INSERT INTO nessie.ns.table VALUES (...);
df.writeTo("nessie.ns.table").append()

-- Mises à jour
UPDATE nessie.ns.table SET col = val WHERE condition;
DELETE FROM nessie.ns.table WHERE condition;
MERGE INTO cible USING source ON condition WHEN MATCHED THEN ... WHEN NOT MATCHED
THEN ...;
```

Voyage dans le temps

```
-- Par identifiant de snapshot
SELECT * FROM table VERSION AS OF 1234567890;

-- Par horodatage
SELECT * FROM table TIMESTAMP AS OF '2024-04-01 12:00:00';

-- Tables de métadonnées
SELECT * FROM table.snapshots;
SELECT * FROM table.history;
SELECT * FROM table.files;
SELECT * FROM table.manifests;
```

Versionnement Nessie

```
CREATE BRANCH nom_branche IN nessie;
USE REFERENCE nom_branche IN nessie;
MERGE BRANCH nom_branche INTO main IN nessie;
DROP BRANCH nom_branche IN nessie;
```

Prochaines étapes recommandées

Pour approfondir les compétences acquises dans ce chapitre, voici des pistes d'exploration :

Expérimentation locale - Augmentez le volume de données générées pour observer le comportement d'Iceberg à plus grande échelle - Testez différentes stratégies de partitionnement et comparez les performances de requête - Explorez les opérations de maintenance (compaction, expiration des snapshots) couvertes au Chapitre IV.10

Intégration Kafka - Configurez un connecteur Kafka vers Iceberg pour l'ingestion en continu (référence : Volume III) - Implémentez un pipeline de streaming lakehouse bout en bout

Exploration avancée de Dremio - Testez les capacités de fédération avec une base de données PostgreSQL locale - Expérimentez avec les différents types de réflexions (raw, aggregation) - Explorez l'API REST de Dremio pour l'automatisation

Comparaison des approches d'interrogation

Le tableau suivant résume les caractéristiques des différentes méthodes d'accès aux données Iceberg :

Approche	Cas d'utilisation	Avantages	Limites
Spark SQL direct	ETL, transformations complexes	Contrôle total, toutes les fonctionnalités Iceberg	Nécessite compétences Spark
Dremio SQL	Analyse ad hoc, exploration	Interface simple, accélération automatique	Dépendance à Dremio
BI via Dremio	Tableaux de bord, rapports	Accessible aux analystes métier	Personnalisation limitée
API Iceberg (Python/Java)	Automatisation, scripts	Flexibilité maximale	Courbe d'apprentissage

Transition vers la Partie 2

Ce chapitre conclut la Partie 1 « La Valeur du Lakehouse ». Vous disposez maintenant d'une compréhension solide des concepts fondamentaux et d'une expérience pratique avec les outils clés de l'écosystème.

La Partie 2 « Concevoir Votre Architecture » vous guidera dans la conception d'une architecture Lakehouse de production. Nous aborderons méthodiquement chaque couche : stockage, ingestion, catalogue, fédération et consommation. Le Chapitre IV.4 débutera par l'audit de votre plateforme existante et la définition des exigences architecturales, illustrés par l'étude de cas détaillée de la Banque Hamerliwa.

Nettoyage de l'environnement de développement

Lorsque vous avez terminé vos expérimentations, vous pouvez nettoyer l'environnement Docker de plusieurs façons selon vos besoins :

Conservation des données pour une session ultérieure :

```
# Arrêter les conteneurs sans supprimer les volumes
docker-compose stop
```

Redémarrage avec les données préservées :

```
docker-compose start
```

Suppression complète (conteneurs et volumes) :

```
# Arrêter et supprimer les conteneurs
docker-compose down

# Supprimer également les volumes (perte des données)
docker-compose down -v

# Nettoyer les images téléchargées (optionnel)
docker image prune -a
```

Cette gestion flexible de l'environnement illustre l'un des avantages de la conteneurisation : la capacité de créer, détruire et recréer des environnements complets de manière reproductible.

> **Migration**
> *De* : Environnement de développement local (Docker Compose)
> *Vers* : Architecture de production infonuagique ou hybride
> *Stratégie* : Les chapitres de la Partie 2 couvrent la sélection des composants, le dimensionnement, la haute disponibilité et les considérations de coûts pour chaque couche de l'architecture. L'approche privilégie une migration progressive, couche par couche, minimisant les risques et permettant une validation incrémentale. Les patrons architecturaux établis dans ce chapitre — séparation stockage/calcul, catalogue centralisé, couche de fédération — se transposent directement vers les déploiements de production, que ce soit sur Amazon Web Services, Google Cloud Platform, Microsoft Azure ou des infrastructures sur site.

Ressources complémentaires

Pour approfondir les sujets abordés dans ce chapitre, les ressources suivantes sont recommandées :

Documentation officielle - Documentation Apache Iceberg : https://iceberg.apache.org/docs/ - Guide Dremio : https://docs.dremio.com/ - Projet Nessie : https://projectnessie.org/

Formations et tutoriels - Dremio University (formations gratuites) : https://www.dremio.com/university/ - Tutoriels Iceberg de la communauté : https://iceberg.apache.org/community/

Livres - « Apache Iceberg: The Definitive Guide » (O'Reilly, 2024) — référence complète sur le format de table

Références

- Apache Iceberg Documentation (2025). *Spark Configuration*. https://iceberg.apache.org/docs/latest/spark-configuration/
- Apache Iceberg (2025). *Release Notes 1.9.x*. https://iceberg.apache.org/releases/
- Dremio (2024). *Apache Iceberg Integration Guide*. https://docs.dremio.com/
- Dremio (2024). *2024 Year in Review: Lakehouses, Apache Iceberg and Dremio*. https://www.dremio.com/blog/
- Project Nessie (2024). *Nessie and Apache Iceberg*. https://projectnessie.org/
- MinIO (2025). *The Definitive Guide to Lakehouse Architecture with Iceberg and MinIO*. https://blog.min.io/
- Apache Software Foundation (2025). *Apache Superset Documentation*. https://superset.apache.org/docs/

Chapitre IV.4 — Préparer Votre Passage à Apache Iceberg

La Partie 1 de ce volume vous a présenté la valeur du Data Lakehouse, l'anatomie technique d'Apache Iceberg et une expérience pratique avec les outils de l'écosystème. Vous disposez désormais d'une compréhension solide des concepts fondamentaux et d'une intuition concrète sur le fonctionnement d'Iceberg. La Partie 2 marque une transition cruciale : nous passons de l'exploration technologique à la conception architecturale pour votre organisation.

Toute migration vers une nouvelle architecture de données représente un investissement significatif en temps, en ressources et en capital humain. Les échecs de projets de modernisation des données sont malheureusement fréquents, et leurs causes sont rarement techniques. Dans la grande majorité des cas, ces échecs résultent d'une préparation insuffisante : une compréhension incomplète de l'existant, des exigences mal définies, ou une communication déficiente avec les parties prenantes.

Ce chapitre établit les fondations méthodologiques pour votre passage à Apache Iceberg. Nous aborderons d'abord la réalisation d'un audit exhaustif de votre plateforme de données actuelle, puis nous illustrerons cette démarche à travers l'étude de cas détaillée de la Banque Hamerliwa, une institution financière fictive mais représentative des défis rencontrés par les organisations canadiennes. Ensuite, nous transformerons les constats de l'audit en exigences architecturales structurées, et finalement, nous explorerons les stratégies de communication et de présentation pour obtenir l'adhésion des décideurs.

L'approche préconisée privilégie la rigueur méthodologique sans sacrifier la pragmatisme. Un audit trop superficiel manquera des éléments critiques qui resurgiront pendant l'implémentation ; un audit excessivement détaillé paralysera le projet avant même qu'il ne commence. L'objectif est de trouver le juste équilibre entre exhaustivité et vélocité, entre analyse et action.

IV.4.1 Réalisation de l'audit de votre plateforme

L'audit de la plateforme de données constitue la première étape indispensable de tout projet de migration vers Apache Iceberg. Cette démarche systématique vise à établir une cartographie complète de l'existant, à identifier les forces et les faiblesses de l'architecture actuelle, et à révéler les opportunités d'amélioration que le Lakehouse peut adresser. Sans cette compréhension approfondie, les décisions architecturales reposeront sur des hypothèses plutôt que sur des faits, augmentant considérablement les risques d'échec.

Objectifs de l'audit

Un audit de plateforme de données bien conduit poursuit plusieurs objectifs complémentaires qui, ensemble, fournissent la base informationnelle nécessaire aux décisions architecturales.

Le premier objectif consiste à dresser l'inventaire complet des actifs de données. Cet inventaire couvre les sources de données (systèmes opérationnels, applications, flux externes), les zones de stockage (bases de

données, entrepôts, lacs de données), les pipelines de transformation (processus ETL, traitements par lots, flux temps réel) et les outils de consommation (tableaux de bord, rapports, modèles analytiques). Cette cartographie révèle souvent des surprises : des sources de données oubliées, des pipelines redondants, ou des dépendances non documentées.

Le deuxième objectif vise à évaluer la santé technique de la plateforme. Cette évaluation examine les performances actuelles (temps de traitement, latence des requêtes, disponibilité), la qualité des données (complétude, exactitude, cohérence), la dette technique accumulée (code obsolète, dépendances désuètes, documentation manquante) et les risques opérationnels (points de défaillance unique, procédures de reprise non testées).

Le troisième objectif cherche à comprendre les besoins métier non satisfaits. Les limitations de la plateforme actuelle créent des frustrations chez les utilisateurs : rapports trop lents, données périmées, impossibilité de croiser certaines sources, coûts prohibitifs pour de nouvelles analyses. Ces besoins non comblés représentent la justification métier de la migration vers le Lakehouse.

Le quatrième objectif établit la base de référence pour mesurer le succès de la migration. Sans métriques initiales documentées, il sera impossible de démontrer objectivement les améliorations apportées par la nouvelle architecture.

Les six dimensions de l'audit

Pour structurer la collecte d'information, nous proposons un cadre d'audit organisé en six dimensions complémentaires. Chaque dimension explore un aspect spécifique de la plateforme et contribue à la vision d'ensemble.

Dimension 1 : Architecture et infrastructure

Cette dimension cartographie les composants techniques de la plateforme actuelle. L'audit documente les systèmes de stockage utilisés (bases relationnelles, systèmes de fichiers distribués, stockage objet), les moteurs de traitement déployés (Spark, Hive, moteurs propriétaires), les outils d'orchestration (Airflow, Control-M, solutions maison) et l'infrastructure sous-jacente (sur site, infonuagique, hybride).

Questions clés à explorer : - Quels sont les principaux systèmes de stockage et leurs volumes respectifs ? - Quels moteurs de calcul sont utilisés pour quels types de traitements ? - Comment l'infrastructure est-elle dimensionnée et quels sont les goulots d'étranglement ? - Quelle est la répartition entre ressources sur site et infonuagiques ? - Quels sont les accords de niveau de service (SLA) actuels et sont-ils respectés ?

Dimension 2 : Données et modélisation

Cette dimension examine la structure et l'organisation des données. L'audit inventorie les schémas de données, les modèles dimensionnels, les zones de données (brutes, raffinées, agrégées) et les relations entre les ensembles de données. Une attention particulière est portée aux formats de fichiers utilisés, aux stratégies de partitionnement en place et aux mécanismes de gestion des schémas.

Questions clés à explorer : - Quels formats de fichiers sont utilisés (Parquet, ORC, Avro, CSV, JSON) ? - Comment les données sont-elles partitionnées et cette stratégie est-elle optimale ? - Existe-t-il un catalogue de métadonnées centralisé ? - Comment l'évolution des schémas est-elle gérée aujourd'hui ? - Quelle est la volumétrie par zone de données et son taux de croissance ?

Dimension 3 : Flux et pipelines

Cette dimension analyse les processus de mouvement et de transformation des données. L'audit recense les pipelines d'ingestion (par lots et en continu), les transformations ETL/ELT, les dépendances entre

pipelines et les fenêtres de traitement. L'objectif est de comprendre comment les données circulent à travers la plateforme, du point d'origine jusqu'à la consommation.

Questions clés à explorer : - Combien de pipelines sont en production et quelle est leur criticité ? - Quels sont les temps de traitement moyens et leurs variations ? - Existe-t-il des pipelines en temps réel et quelles technologies utilisent-ils ? - Comment les erreurs et les reprises sont-elles gérées ? - Quelle est la fréquence de rafraîchissement des différentes zones de données ?

Dimension 4 : Consommation et utilisateurs

Cette dimension évalue comment les données sont utilisées par les différentes communautés d'utilisateurs. L'audit identifie les outils de consommation (outils BI, notebooks, applications), les profils d'utilisateurs (analystes, scientifiques de données, développeurs, dirigeants), les cas d'usage principaux et les volumes de requêtes.

Questions clés à explorer : - Quels outils BI sont utilisés et par combien d'utilisateurs ? - Quelles sont les requêtes les plus fréquentes et les plus coûteuses ? - Existe-t-il des besoins d'analyse ad hoc non satisfaits ? - Comment les utilisateurs accèdent-ils aux données (SQL, API, fichiers) ? - Quels sont les temps de réponse actuels et les attentes des utilisateurs ?

Dimension 5 : Gouvernance et sécurité

Cette dimension examine les mécanismes de contrôle et de protection des données. L'audit évalue les politiques d'accès, les processus de catalogage, la gestion du lignage, la conformité réglementaire et les pratiques de qualité des données.

Questions clés à explorer : - Comment les accès aux données sont-ils contrôlés ? - Existe-t-il un catalogue de données et est-il à jour ? - Comment le lignage des données est-il tracé ? - Quelles réglementations s'appliquent (Loi 25, RGPD, réglementations sectorielles) ? - Quels processus de qualité des données sont en place ?

Dimension 6 : Coûts et opérations

Cette dimension quantifie les aspects économiques et opérationnels de la plateforme. L'audit documente les coûts directs (infrastructure, licences, personnel), les coûts indirects (temps perdu, opportunités manquées), la charge opérationnelle et les incidents récurrents.

Questions clés à explorer : - Quel est le coût total de possession (TCO) de la plateforme actuelle ? - Comment les coûts sont-ils répartis entre stockage, calcul et licences ? - Quelle est la charge de travail de l'équipe de données ? - Quels sont les incidents les plus fréquents et leur impact ? - Existe-t-il des inefficacités connues (données dupliquées, traitements redondants) ?

Méthodes de collecte d'information

La collecte d'information pour l'audit combine plusieurs méthodes complémentaires, chacune apportant une perspective différente sur la plateforme.

Analyse documentaire : L'examen de la documentation existante (diagrammes d'architecture, spécifications techniques, manuels opérationnels) fournit une première vue de la plateforme. Toutefois, la documentation est souvent incomplète ou obsolète, ce qui nécessite une validation par d'autres méthodes. Les documents à collecter incluent les schémas de données, les dictionnaires de données, les procédures opérationnelles, les rapports d'incidents passés, les contrats avec les fournisseurs et les évaluations de sécurité antérieures.

Entretiens structurés : Les conversations avec les différentes parties prenantes révèlent des informations que les documents ne capturent pas : les frustrations quotidiennes, les solutions de contournement,

les besoins non exprimés. Les entretiens doivent couvrir les équipes techniques (administrateurs, développeurs de pipelines, analystes de données) et les utilisateurs métier (analystes financiers, équipes marketing, dirigeants). Un guide d'entretien standardisé assure la cohérence des questions tout en laissant place aux explorations spontanées. Prévoyez 60 à 90 minutes par entretien et documentez les réponses immédiatement pour éviter les pertes d'information.

Analyse technique : L'examen direct des systèmes fournit des données objectives sur la performance, les volumes et les configurations. Cette analyse inclut l'inspection des métadonnées, le profilage des requêtes, l'analyse des journaux système et le recensement des composants installés. Pour les systèmes de données, portez une attention particulière aux statistiques de tables (cardinalité, distribution des valeurs, taux de nullité), aux plans d'exécution des requêtes fréquentes, aux métriques de latence et de débit, ainsi qu'aux alertes et incidents récents dans les outils de supervision.

Ateliers collaboratifs : Les sessions de travail en groupe permettent de valider les constats, de résoudre les ambiguïtés et de construire une compréhension partagée de la situation actuelle. Ces ateliers rassemblent des représentants des différentes équipes concernées. Structurez ces ateliers autour de questions ouvertes comme « Quels sont les trois problèmes de données qui vous font perdre le plus de temps ? » ou « Si vous pouviez changer une chose dans la plateforme actuelle, ce serait quoi ? ». Utilisez des techniques de facilitation comme le vote par points pour prioriser les enjeux identifiés.

Observation directe : Passez du temps avec les équipes pendant leurs activités quotidiennes. Observez comment les analystes accèdent aux données, quelles manipulations manuelles ils effectuent, combien de temps ils attendent les résultats de leurs requêtes. Cette méthode révèle des inefficacités que les personnes concernées ne mentionnent plus parce qu'elles les considèrent comme normales.

> **Étude de cas : Banque TD — L'audit qui a révélé les angles morts**
> *Secteur* : Services financiers
> *Défi* : Un audit initial basé uniquement sur la documentation existante avait conclu que la plateforme de données était en bon état. Les projets de modernisation basés sur ces conclusions ont échoué.
> *Solution* : Un second audit incorporant des entretiens approfondis et une observation directe des équipes a révélé que 40 % des traitements critiques reposaient sur des scripts non documentés maintenus par des individus spécifiques.
> *Résultats* : La cartographie complète des dépendances cachées a permis de planifier une migration réaliste, complétée avec 3 mois d'avance sur le calendrier révisé.

Livrables de l'audit

L'audit produit plusieurs livrables qui serviront de référence tout au long du projet de migration. La qualité et la complétude de ces livrables déterminent directement la qualité des décisions architecturales qui en découleront.

Le **rapport d'audit** synthétise les constats organisés par dimension, accompagnés d'une évaluation de la maturité de la plateforme et d'une identification des risques et opportunités. Ce document de 30 à 50 pages constitue la référence principale pour les décideurs. Il doit être rédigé dans un langage accessible aux non-techniciens tout en conservant la rigueur nécessaire pour les experts. Une version exécutive de 5 pages accompagne le rapport complet pour les audiences pressées.

La **cartographie des actifs** présente visuellement l'architecture actuelle : systèmes, flux de données, dépendances et zones de responsabilité. Cette cartographie utilise idéalement une notation standardisée (comme ArchiMate ou C4) pour faciliter la communication et la maintenance. Elle doit pouvoir être mise à jour au fil du projet pour refléter l'évolution de l'architecture.

L'**inventaire des données** liste les ensembles de données avec leurs caractéristiques : volume, format, propriétaire, niveau de sensibilité, fréquence d'utilisation, qualité observée. Pour les grandes organisations, cet inventaire peut contenir des milliers d'entrées. Une priorisation basée sur la criticité métier et la complexité technique aide à focaliser les efforts de migration.

Le **catalogue des pipelines** documente les processus de données : sources, transformations, destinations, planification, propriétaires, SLA associés. Ce catalogue identifie également les dépendances entre pipelines, information critique pour planifier l'ordre de migration.

Les **métriques de référence** établissent les indicateurs de performance actuels qui serviront de base de comparaison après la migration. Ces métriques couvrent les dimensions de performance (temps de traitement, latence), de qualité (taux d'erreurs, complétude), de coût (par téraoctet, par requête) et de satisfaction (tickets de support, temps de résolution).

Le **registre des risques** identifie les risques découverts pendant l'audit avec leur probabilité, leur impact et les stratégies de mitigation proposées. Ce registre sera enrichi tout au long du projet.

> **Performance**
> L'audit lui-même peut révéler des opportunités d'amélioration rapide (« quick wins ») indépendantes de la migration Iceberg. Par exemple, l'identification de requêtes mal optimisées, de partitionnements inadéquats ou de données dupliquées peut justifier des corrections immédiates qui amélioreront la situation sans attendre le déploiement du Lakehouse. Chez Hamerliwa, l'audit a identifié 8 quick wins potentiels représentant une économie annuelle de 180 K$ et une amélioration de 25 % des temps de traitement nocturnes.

Facteurs de succès de l'audit

Plusieurs facteurs contribuent à la réussite de l'audit et à l'utilité de ses résultats. L'expérience des projets similaires permet d'identifier les conditions qui maximisent les chances de succès.

Engagement de la direction : L'audit nécessite l'accès à des ressources (temps des équipes, systèmes de production) qui requiert un soutien explicite de la direction. Sans ce soutien, les équipes percevront l'audit comme une distraction de leurs responsabilités quotidiennes. Le sponsor exécutif doit communiquer clairement l'importance de l'audit et autoriser explicitement les équipes à y consacrer le temps nécessaire.

Périmètre réaliste : Un audit trop ambitieux s'enlisera dans les détails ; un audit trop restreint manquera des éléments critiques. La définition du périmètre doit équilibrer exhaustivité et pragmatisme en fonction des ressources disponibles. En règle générale, prévoyez 2 à 3 jours d'effort par dimension pour une organisation de taille moyenne comme Hamerliwa.

Équipe pluridisciplinaire : L'audit bénéficie de la diversité des perspectives. Une équipe composée d'architectes, d'ingénieurs de données, d'analystes métier et de représentants des opérations couvrira mieux l'ensemble des dimensions. L'inclusion d'un regard externe (consultant, auditeur) apporte une objectivité que les équipes internes, trop proches des systèmes, peuvent avoir du mal à maintenir.

Communication transparente : L'audit n'est pas une mission d'inspection visant à identifier des coupables. Cette distinction doit être clairement communiquée pour obtenir la coopération des équipes et des informations franches sur les problèmes existants. Les participants doivent comprendre que l'objectif est d'améliorer la situation, pas de pointer du doigt les responsables des problèmes passés.

Calendrier réaliste : Un audit bâclé en deux semaines manquera des éléments essentiels. À l'inverse, un audit qui s'étire sur six mois perdra son élan et ses conclusions risquent d'être déjà obsolètes. Pour une organisation de la taille d'Hamerliwa, un calendrier de 6 à 8 semaines représente un bon équilibre.

Accès aux données réelles : L'audit doit pouvoir examiner les systèmes de production, pas seulement la documentation. Les métriques de performance, les journaux d'erreurs, les statistiques d'utilisation fournissent une image objective que la documentation seule ne peut pas donner. Prévoyez les autorisations nécessaires dès le début du projet.

Documentation au fil de l'eau : Documenter les constats immédiatement, pendant que le contexte est frais, améliore considérablement la qualité du rapport final. Attendre la fin de l'audit pour tout documenter entraîne des pertes d'information et des incohérences.

IV.4.2 L'audit de la Banque Hamerliwa en action

Pour illustrer concrètement la démarche d'audit, nous suivrons le cas de la Banque Hamerliwa, une institution financière canadienne de taille moyenne. Bien que fictive, cette banque représente un archétype réaliste des organisations qui envisagent la migration vers un Lakehouse Apache Iceberg. Son histoire, ses défis et ses aspirations reflètent les situations rencontrées dans de nombreuses entreprises canadiennes du secteur financier et au-delà.

Profil de la Banque Hamerliwa

La Banque Hamerliwa est une banque régionale établie depuis 1952, dont le siège social se situe à Québec. Avec 2 500 employés répartis dans 85 succursales au Québec et en Ontario, elle gère 45 milliards de dollars d'actifs et sert 850 000 clients particuliers et 35 000 entreprises. Son nom, signifiant « banque des gens » en langue algonquine, reflète sa mission historique d'accessibilité financière.

Au cours de la dernière décennie, Hamerliwa a connu plusieurs transformations technologiques successives. En 2015, elle a migré son système bancaire central vers une plateforme moderne. En 2018, elle a lancé une application mobile qui a rapidement gagné en popularité. En 2020, face à la pandémie, elle a accéléré sa transformation numérique. Chacune de ces initiatives a généré de nouveaux besoins en données, aboutissant à une architecture complexe et parfois incohérente.

Contexte de l'audit

En septembre 2025, le Conseil d'administration d'Hamerliwa approuve un programme de modernisation des données sur trois ans, doté d'un budget de 28 millions de dollars. Ce programme répond à plusieurs pressions convergentes :

Pression réglementaire : L'entrée en vigueur complète de la Loi 25 au Québec et les exigences accrues du BSIF (Bureau du surintendant des institutions financières) en matière de gestion des données imposent des améliorations dans la gouvernance et la traçabilité.

Pression concurrentielle : Les néobanques et les fintechs offrent des expériences client personnalisées basées sur l'analyse avancée des données. Hamerliwa doit développer des capacités similaires pour maintenir sa position sur le marché.

Pression opérationnelle : La plateforme de données actuelle montre des signes de fatigue. Les rapports réglementaires prennent de plus en plus de temps à produire, les équipes d'analyse passent davantage de

temps à préparer les données qu'à les analyser, et les coûts d'infrastructure augmentent plus vite que les volumes de données.

L'équipe d'audit est constituée en octobre 2025, sous la direction de Maryse Chen, architecte de données senior. Elle est composée de deux ingénieurs de données, d'une analyste métier, d'un spécialiste de la gouvernance et d'un consultant externe spécialisé dans les architectures Lakehouse. L'audit doit être complété en huit semaines.

Constats de l'audit — Dimension Architecture

L'analyse de l'architecture révèle une plateforme fragmentée, résultat de décisions prises à différentes époques pour répondre à des besoins spécifiques.

Systèmes de stockage multiples : Hamerliwa exploite simultanément un entrepôt de données Oracle Exadata acquis en 2014, un cluster Hadoop Cloudera déployé en 2017 pour le « big data », une instance Snowflake adoptée en 2021 pour les analyses marketing, et plusieurs bases de données Azure SQL pour des applications départementales. Cette multiplicité génère des silos de données, des duplications et des incohérences.

Infrastructure hybride non optimisée : L'Exadata reste dans le centre de données principal de Québec, le cluster Hadoop occupe deux armoires vieillissantes, tandis que Snowflake et Azure SQL fonctionnent dans l'infonuagique. Cette répartition résulte davantage de l'historique que d'une stratégie délibérée. Les coûts de transfert de données entre environnements sont significatifs.

Moteurs de calcul disparates : Les traitements s'exécutent sur Informatica PowerCenter (ETL traditionnel), Apache Spark sur Hadoop (traitements big data), Snowflake (analyses marketing) et des scripts Python ad hoc. Chaque moteur a ses experts, ses pratiques et sa dette technique.

Composant	Année	Volumes	Coût annuel
Oracle Exadata	2014	15 To	1,2 M$
Cloudera Hadoop	2017	180 To	850 K$
Snowflake	2021	8 To	420 K$
Azure SQL (divers)	2019-2023	2 To	180 K$

Constats de l'audit — Dimension Données

L'examen des données révèle des pratiques de gestion hétérogènes et une dette technique substantielle.

Formats multiples sans standard : Les données sont stockées en format CSV (systèmes hérités), Parquet (traitements Spark récents), ORC (premiers projets Hadoop), tables relationnelles (Exadata, SQL), et formats propriétaires Snowflake. Cette diversité complique les traitements inter-systèmes.

Partitionnement incohérent : Sur Hadoop, certaines tables sont partitionnées par date, d'autres par région, d'autres encore ne sont pas partitionnées du tout. Les choix de partitionnement ne correspondent pas toujours aux patterns d'accès, entraînant des analyses de données complètes (« full scans ») évitables.

Évolution de schéma problématique : Les modifications de schéma sur Hadoop nécessitent souvent une réécriture complète des données historiques. Plusieurs tables contiennent des colonnes obsolètes conservées par prudence, tandis que d'autres manquent de colonnes ajoutées récemment dans les systèmes sources.

Catalogue incomplet : Un outil de catalogage a été déployé en 2022, mais seulement 35 % des actifs de données y sont documentés. Les métadonnées sont souvent périmées, et le lignage n'est tracé que pour les pipelines les plus critiques.

Constats de l'audit — Dimension Flux

L'analyse des pipelines met en lumière une complexité accumulée au fil des ans.

Prolifération des pipelines : L'inventaire recense 847 pipelines en production, dont 312 considérés comme critiques. Cependant, 23 % des pipelines n'ont pas été modifiés depuis plus de trois ans et leur pertinence est incertaine. Certains pipelines font double emploi, produisant des résultats similaires par des chemins différents.

Fenêtres de traitement saturées : Les traitements nocturnes par lots s'étendent désormais de 22h à 7h, ne laissant qu'une marge de 3 heures avant l'ouverture des marchés. Tout retard significatif impacte la disponibilité des rapports matinaux.

Absence de temps réel : Malgré l'acquisition d'une licence Kafka en 2023, les flux temps réel restent limités à quelques cas d'usage pilotes. La majorité des données analytiques ont un délai de fraîcheur de T+1 (disponibles le lendemain).

Gestion des erreurs artisanale : Les reprises après erreur sont largement manuelles. L'équipe des opérations consacre en moyenne 15 heures par semaine à la correction de pipelines échoués.

> **Étude de cas : Banque Hamerliwa — Le syndrome du pipeline orphelin**
> *Secteur* : Services financiers
> *Défi* : Au cours de l'audit, l'équipe découvre 47 pipelines dont les propriétaires ont quitté l'organisation sans transfert de connaissances. Ces pipelines continuent de s'exécuter, consommant des ressources, mais personne ne peut confirmer si leurs résultats sont encore utilisés.
> *Solution* : Un processus de « quarantaine » est mis en place : les pipelines suspects sont désactivés temporairement et, en l'absence de réclamation après 30 jours, ils sont archivés.
> *Résultats* : 31 pipelines sont définitivement retirés, libérant 8 % de la capacité de calcul Hadoop et simplifiant la planification nocturne.

Constats de l'audit — Dimension Consommation

L'évaluation de l'utilisation des données révèle un écart entre les capacités offertes et les besoins réels. Cette dimension est particulièrement importante car elle détermine la valeur perçue de la plateforme de données par les utilisateurs métier.

Outils BI multiples : Hamerliwa utilise simultanément Power BI (équipes centrales, 180 utilisateurs), Tableau (équipes marketing, 45 utilisateurs), QlikView (hérité, encore utilisé par la comptabilité, 30 utilisateurs) et des rapports Excel personnalisés (nombre indéterminé mais estimé à plusieurs centaines). Cette diversité complique la gouvernance et génère des résultats parfois contradictoires pour les mêmes indicateurs. L'audit identifie 12 cas où le même KPI est calculé différemment selon l'outil utilisé.

Performance insuffisante : Les requêtes analytiques complexes sur l'Exadata peuvent prendre plusieurs heures. L'audit mesure un temps de réponse médian de 47 minutes pour les requêtes impliquant des jointures entre plus de 5 tables. Les analystes contournent cette limitation en extrayant des échantillons de données vers des fichiers locaux, créant des risques de sécurité et de cohérence. On estime que 2,3 To de données « grises » existent sur les postes de travail et les espaces de partage non gouvernés.

Besoins non satisfaits : Les entretiens avec les équipes métier révèlent des frustrations récurrentes documentées dans le tableau suivant :

Équipe	Besoin exprimé	Impact estimé
Gestion des risques	Analyses temps réel pour détection de fraude	2,1 M$ de pertes évitables/an
Marketing	Segmentation avancée avec données comportementales	15 % d'amélioration potentielle des campagnes
Finance	Clôtures accélérées avec réconciliation automatique	5 jours/mois récupérés
Direction	Tableaux de bord consolidés multi-domaines	Décisions plus rapides et mieux informées
Conformité	Rapports réglementaires automatisés	Réduction des erreurs de 40 %

Compétences limitées : Seuls 12 employés maîtrisent SQL avancé, créant un goulot d'étranglement pour les demandes d'analyse. Les scientifiques de données (équipe de 8 personnes) passent 60 % de leur temps à préparer les données plutôt qu'à construire des modèles. Le ratio préparation/analyse devrait idéalement être inversé. Cette situation freine l'adoption de l'apprentissage automatique malgré les investissements réalisés dans les outils.

Libre-service limité : Les tentatives de déployer des outils de libre-service analytique ont échoué faute de données suffisamment préparées et documentées. Les utilisateurs métier abandonnent rapidement lorsqu'ils ne trouvent pas les données dont ils ont besoin ou ne comprennent pas leur signification.

Constats de l'audit — Dimension Gouvernance

L'examen de la gouvernance expose des lacunes qui deviennent critiques face aux exigences réglementaires croissantes.

Contrôle d'accès fragmenté : Chaque système possède son propre mécanisme de gestion des accès. Un employé peut avoir accès à des données sensibles dans Snowflake sans autorisation équivalente dans l'Exadata, créant des incohérences difficiles à auditer.

Lignage incomplet : Le lignage des données n'est documenté que pour 40 % des pipelines critiques. En cas de question réglementaire sur l'origine d'une donnée, la reconstitution manuelle peut prendre plusieurs jours.

Qualité des données variable : Des règles de qualité existent pour les données réglementaires, mais la majorité des données analytiques ne font l'objet d'aucun contrôle systématique. Les utilisateurs découvrent les problèmes de qualité lors de l'analyse, générant perte de temps et méfiance.

Conformité Loi 25 : L'évaluation révèle des écarts par rapport aux exigences de la Loi 25, notamment en matière de consentement, de conservation et de droit à l'effacement. Les données personnelles sont dispersées dans plusieurs systèmes, compliquant la réponse aux demandes d'accès.

Constats de l'audit — Dimension Coûts

L'analyse financière quantifie le fardeau économique de l'architecture actuelle.

Coût total de possession : Les coûts directs de la plateforme de données s'élèvent à 4,8 millions de dollars annuellement, répartis entre infrastructure (2,65 M), $licenceslogicielles(1,35M)$ et services

professionnels (0,8 M$). Ces coûts ont augmenté de 18 % sur les deux dernières années, principalement à cause de Snowflake et des transferts de données inter-environnements.

Coûts cachés : Les coûts indirects sont plus difficiles à quantifier mais substantiels. L'équipe estime à 1,2 million de dollars la valeur du temps perdu par les analystes en préparation de données. Les opportunités manquées faute de capacités analytiques adéquates sont incalculables.

Structure de coûts rigide : 78 % des coûts sont fixes (contrats pluriannuels, infrastructure sur site). Cette rigidité limite la capacité d'adaptation aux variations de la demande.

Catégorie	Coût annuel	% du total	Tendance
Infrastructure sur site	1,45 M$	30 %	Stable
Infonuagique (Snowflake, Azure)	1,20 M$	25 %	+25 %/an
Licences (Oracle, Informatica, BI)	1,35 M$	28 %	+5 %/an
Personnel dédié	0,55 M$	12 %	+10 %/an
Services professionnels	0,25 M$	5 %	Variable

Synthèse de l'audit Hamerliwa

L'audit de la Banque Hamerliwa dresse le portrait d'une plateforme de données fonctionnelle mais fragile, dont la complexité et les coûts augmentent plus vite que la valeur générée. Les constats principaux se résument ainsi :

Forces identifiées : - Équipe technique compétente et motivée - Données réglementaires de bonne qualité - Infrastructure Exadata stable pour les traitements critiques - Expérience initiale avec Kafka et technologies modernes

Faiblesses majeures : - Architecture fragmentée en silos technologiques - Absence de catalogue de données centralisé et complet - Incapacité à répondre aux besoins temps réel - Coûts croissants sans amélioration proportionnelle des capacités - Gouvernance inadaptée aux exigences réglementaires actuelles

Opportunités Lakehouse : - Consolidation des données sur une plateforme unifiée - Réduction des coûts par élimination des duplications - Capacités analytiques avancées (temps réel, ML) - Gouvernance intégrée avec contrôle d'accès fin - Évolution de schéma native sans reconstruction

Risques à mitiger : - Résistance au changement des équipes habituées aux outils existants - Complexité de migration des 847 pipelines - Dépendance aux compétences externes pendant la transition - Continuité des services pendant la migration

IV.4.3 De l'audit aux exigences

Les constats de l'audit constituent une mine d'informations, mais ils doivent être transformés en exigences architecturales exploitables pour guider la conception du Lakehouse. Cette traduction représente une étape critique : des exigences mal formulées mèneront à une architecture inadaptée, tandis que des exigences trop vagues laisseront trop de latitude aux interprétations divergentes.

Structure des exigences

Nous organisons les exigences en quatre catégories complémentaires qui, ensemble, définissent le cadre de la future architecture.

Exigences fonctionnelles : Ce que la plateforme doit permettre de faire. Elles décrivent les capacités requises en termes de stockage, traitement, consommation et gouvernance.

Exigences non fonctionnelles : Comment la plateforme doit se comporter. Elles spécifient les niveaux de performance, disponibilité, sécurité et évolutivité attendus.

Exigences de migration : Comment passer de l'existant à la cible. Elles définissent les contraintes de continuité, les priorités de migration et les critères d'acceptation.

Exigences de gouvernance : Comment la plateforme sera encadrée. Elles établissent les politiques de contrôle, les responsabilités et les processus de gestion.

Exigences fonctionnelles pour Hamerliwa

À partir des constats de l'audit, l'équipe formule les exigences fonctionnelles suivantes pour le futur Lakehouse.

EF-01 : Stockage unifié des données
La plateforme doit permettre le stockage de toutes les données analytiques dans un format ouvert et interopérable. Les données actuellement dispersées entre Exadata (analytique), Hadoop, Snowflake et Azure SQL doivent être consolidables sur une infrastructure commune.

Justification audit : Les silos de données actuels génèrent des duplications (estimées à 40 % des volumes), des incohérences et des coûts de transfert élevés.

EF-02 : Évolution de schéma sans interruption
La plateforme doit supporter l'ajout, la suppression et la modification de colonnes sans nécessiter la réécriture des données historiques ni l'interruption des traitements.

Justification audit : Les modifications de schéma actuelles sur Hadoop mobilisent en moyenne 3 jours-personnes et créent des fenêtres d'indisponibilité.

EF-03 : Capacité de requêtes historiques (Time Travel)
La plateforme doit permettre d'interroger l'état des données à n'importe quel moment passé, avec une rétention configurable par table.

Justification audit : Les demandes réglementaires de reconstitution historique nécessitent actuellement des restaurations de sauvegardes coûteuses et lentes.

EF-04 : Ingestion temps réel
La plateforme doit supporter l'ingestion de données en continu avec un délai maximal de 5 minutes entre l'événement source et la disponibilité pour l'analyse.

Justification audit : Les besoins de détection de fraude et de personnalisation en temps réel ne peuvent être satisfaits avec l'architecture T+1 actuelle.

EF-05 : Accès SQL unifié
La plateforme doit exposer toutes les données via une interface SQL standard, accessible depuis les outils BI existants (Power BI, Tableau) sans développement spécifique.

Justification audit : La multiplication des interfaces d'accès (JDBC Oracle, connecteurs Snowflake, scripts Python) complique l'adoption et génère des erreurs.

EF-06 : Catalogue de métadonnées centralisé
La plateforme doit inclure un catalogue automatiquement alimenté répertoriant tous les actifs de données avec leurs métadonnées techniques et métier.

Justification audit : Le catalogue actuel ne couvre que 35 % des actifs et ses informations sont souvent périmées.

EF-07 : Lignage de données automatique
La plateforme doit capturer automatiquement le lignage des données, de l'ingestion à la consommation, traçant les transformations appliquées.

Justification audit : La reconstitution manuelle du lignage pour les audits réglementaires consomme plusieurs jours-personnes par demande.

Exigences non fonctionnelles pour Hamerliwa

Les exigences non fonctionnelles définissent les niveaux de service attendus de la plateforme. Ces exigences sont souvent négligées au profit des fonctionnalités, mais elles déterminent l'expérience réelle des utilisateurs et la viabilité opérationnelle de la solution.

ENF-01 : Performance des requêtes analytiques
Les requêtes analytiques standards (agrégations sur 30 jours de transactions) doivent s'exécuter en moins de 30 secondes pour les utilisateurs BI. Les requêtes ad hoc complexes doivent s'exécuter en moins de 5 minutes.

Métrique actuelle : 45 minutes en moyenne pour les requêtes complexes sur Exadata.

Critères de mesure : Percentile 95 des temps de réponse mesuré sur une semaine glissante pour un échantillon représentatif de requêtes.

ENF-02 : Disponibilité de la plateforme
La plateforme doit garantir une disponibilité de 99,9 % pendant les heures ouvrables (6h-22h, heure de l'Est) et de 99,5 % hors heures ouvrables.

Métrique actuelle : 99,2 % de disponibilité moyenne, avec des incidents récurrents lors des traitements nocturnes.

Implications : 99,9 % correspond à 8,76 heures d'indisponibilité maximale par an pendant les heures ouvrables. Cette cible nécessite une architecture hautement disponible avec basculement automatique.

ENF-03 : Scalabilité du stockage
La plateforme doit supporter une croissance de 50 % des volumes de données annuellement sans dégradation de performance ni intervention architecturale majeure.

Projection : Les volumes actuels de 205 To atteindront 500 To d'ici 2028 selon les tendances observées.

Contrainte technique : Le stockage objet infonuagique répond nativement à cette exigence, contrairement aux systèmes sur site qui nécessitent des acquisitions matérielles planifiées.

ENF-04 : Élasticité du calcul
La capacité de calcul doit pouvoir augmenter de 300 % en moins de 15 minutes pour absorber les pics de charge (clôtures mensuelles, campagnes marketing).

Contrainte actuelle : L'infrastructure fixe actuelle ne permet aucune élasticité.

Justification : Les analyses de clôture trimestrielle génèrent des pics de charge prévisibles mais intenses. Sans élasticité, ces périodes créent des goulots d'étranglement et des retards dans les livrables.

ENF-05 : Temps de récupération après sinistre
En cas de sinistre majeur, la plateforme doit être récupérable avec un RPO (Recovery Point Objective) de 1 heure et un RTO (Recovery Time Objective) de 4 heures.

État actuel : RPO de 24 heures, RTO non testé estimé à 48 heures.

Implications architecturales : Cette exigence nécessite une réplication continue des métadonnées critiques et des snapshots fréquents des tables Iceberg. La stratégie de récupération doit être documentée et testée trimestriellement.

ENF-06 : Coût cible
Le coût total de possession de la nouvelle plateforme ne doit pas dépasser le coût actuel (4,8 M$/an) la première année, et doit diminuer de 15 % d'ici la troisième année malgré l'augmentation des volumes.

Décomposition cible année 3 : - Stockage objet : 0,8 M$ (contre 1,2 M$ actuel pour Snowflake + transferts) - Calcul élastique : 1,2 M$ (variable selon usage) - Licences (Dremio, outils) : 0,9 M$ (contre 1,35 M$ actuel) - Personnel et formation : 0,7 M$ - Services professionnels : 0,5 M$ - **Total année 3** : 4,1 M$ (réduction de 15 %)

ENF-07 : Latence d'ingestion temps réel
Pour les flux désignés comme temps réel, le délai entre l'événement source et la disponibilité de la donnée pour interrogation ne doit pas dépasser 5 minutes dans 99 % des cas.

Cas d'usage : Détection de fraude, alertes de seuil, tableaux de bord opérationnels.

Contrainte technique : Cette exigence oriente vers une architecture Streaming Lakehouse avec ingestion Kafka directe dans les tables Iceberg, plutôt qu'un modèle micro-batch traditionnel.

ENF-08 : Concurrence utilisateurs
La plateforme doit supporter simultanément 200 utilisateurs BI actifs et 50 requêtes analytiques lourdes sans dégradation significative (< 20 % d'augmentation du temps de réponse).

État actuel : Au-delà de 80 utilisateurs simultanés, les performances de l'Exadata se dégradent notablement.

Exigences de migration pour Hamerliwa

Les exigences de migration encadrent la transition de l'architecture actuelle vers le Lakehouse cible.

EM-01 : Continuité des services critiques
Les services réglementaires (rapports BSIF, déclarations fiscales) doivent rester opérationnels sans interruption tout au long de la migration. Une période de fonctionnement parallèle est requise avant toute décommission.

EM-02 : Migration progressive par domaine
La migration doit procéder par domaine de données (risque, finance, marketing, opérations) plutôt que par technologie, permettant une validation métier à chaque étape.

EM-03 : Réversibilité
Chaque phase de migration doit préserver la capacité de retour à l'état précédent pendant 90 jours, avec des procédures documentées et testées.

EM-04 : Validation des données migrées
Des mécanismes de réconciliation automatique doivent confirmer l'intégrité des données migrées. Une tolérance de 0,01 % est acceptée pour les écarts numériques dus aux conversions de formats.

EM-05 : Transfert de connaissances
La migration doit inclure un programme de formation pour les équipes internes, avec l'objectif que 80 % des opérations courantes soient gérées en interne à la fin du projet.

> **Migration**
> *De* : Architecture fragmentée (Exadata + Hadoop + Snowflake + Azure SQL)
> *Vers* : Lakehouse unifié Apache Iceberg
> *Stratégie* : Migration par domaine avec période de fonctionnement parallèle. Priorité 1 : données marketing (moins critiques, bon terrain d'apprentissage). Priorité 2 : données opérationnelles. Priorité 3 : données réglementaires (après validation approfondie). Décommission des anciens systèmes uniquement après 6 mois de stabilité sur la nouvelle plateforme.

Exigences de gouvernance pour Hamerliwa

Les exigences de gouvernance établissent le cadre de contrôle de la future plateforme.

EG-01 : Contrôle d'accès fin
La plateforme doit supporter des politiques d'accès au niveau des lignes et des colonnes, permettant de restreindre l'accès aux données sensibles selon le profil de l'utilisateur.

Contrainte réglementaire : Loi 25, exigences BSIF sur la séparation des fonctions.

EG-02 : Chiffrement des données
Toutes les données doivent être chiffrées au repos (AES-256) et en transit (TLS 1.3). Les clés de chiffrement doivent être gérées par un service dédié avec rotation automatique.

EG-03 : Conservation et purge
La plateforme doit supporter des politiques de conservation configurables par classification de données, avec purge automatique à l'expiration et preuve de destruction.

EG-04 : Audit des accès
Tous les accès aux données sensibles doivent être journalisés avec horodatage, identité de l'utilisateur, action effectuée et données consultées. Ces journaux doivent être conservés 7 ans.

EG-05 : Propriété des données
Chaque ensemble de données doit avoir un propriétaire métier identifié, responsable de la qualité, des accès et du cycle de vie.

Matrice de traçabilité

Pour assurer que chaque exigence répond à un constat de l'audit et que chaque constat critique est adressé, une matrice de traçabilité est établie :

Constat audit	Exigences associées	Priorité
Silos de données multiples	EF-01, EF-05	Critique
Évolution de schéma difficile	EF-02	Haute
Pas de capacité temps réel	EF-04	Haute
Catalogue incomplet	EF-06	Moyenne
Lignage non tracé	EF-07	Haute
Requêtes lentes	ENF-01	Critique
Coûts croissants	ENF-06	Haute
Gouvernance fragmentée	EG-01 à EG-05	Critique

Cette matrice sera utilisée tout au long du projet pour vérifier que l'architecture conçue répond effectivement aux besoins identifiés.

IV.4.4 Plan architectural et présentation itinérante

Les exigences définies à la section précédente constituent le cahier des charges de la future architecture. Avant d'entamer la conception détaillée — qui sera l'objet des chapitres IV.5 à IV.9 — il est essentiel d'établir un plan architectural de haut niveau et de le communiquer efficacement aux parties prenantes. Cette section traite de ces deux aspects complémentaires : la vision architecturale et la stratégie de communication.

Vision architecturale de haut niveau

Le plan architectural traduit les exigences en orientations technologiques et organisationnelles. Pour Hamerliwa, la vision architecturale s'articule autour de cinq piliers.

Pilier 1 : Apache Iceberg comme format de table universel

Apache Iceberg sera adopté comme format de table unique pour toutes les données analytiques. Ce choix répond aux exigences EF-01 (stockage unifié), EF-02 (évolution de schéma), EF-03 (Time Travel) et apporte plusieurs bénéfices :

- Format ouvert évitant le verrouillage fournisseur
- Support natif de l'évolution de schéma et du partitionnement masqué
- Interopérabilité avec tous les moteurs de calcul majeurs
- Snapshots pour le Time Travel et la récupération après erreur

Pilier 2 : Stockage objet infonuagique

Les données seront stockées sur un service de stockage objet infonuagique (à déterminer : Azure Blob Storage, Amazon S3, ou Google Cloud Storage selon l'analyse du chapitre IV.5). Cette orientation répond aux exigences ENF-03 (scalabilité), ENF-04 (élasticité) et ENF-06 (coûts) :

- Scalabilité pratiquement illimitée
- Modèle de coûts à l'usage aligné sur la consommation réelle

- Durabilité et disponibilité natives
- Séparation stockage/calcul pour optimisation des coûts

Pilier 3 : Catalogue REST unifié

Un catalogue conforme à la spécification Apache Iceberg REST Catalog centralisera la gestion des métadonnées. Cette orientation répond aux exigences EF-06 (catalogue centralisé) et EF-07 (lignage) :

- Point d'accès unique pour tous les moteurs de calcul
- Versionnement des métadonnées (capacité Git-like)
- Intégration native avec les outils de gouvernance
- Support du multi-tenancy pour les environnements isolés

Pilier 4 : Couche de fédération Dremio

Dremio sera déployé comme couche de fédération et d'accélération, répondant aux exigences EF-05 (accès SQL unifié) et ENF-01 (performance) :

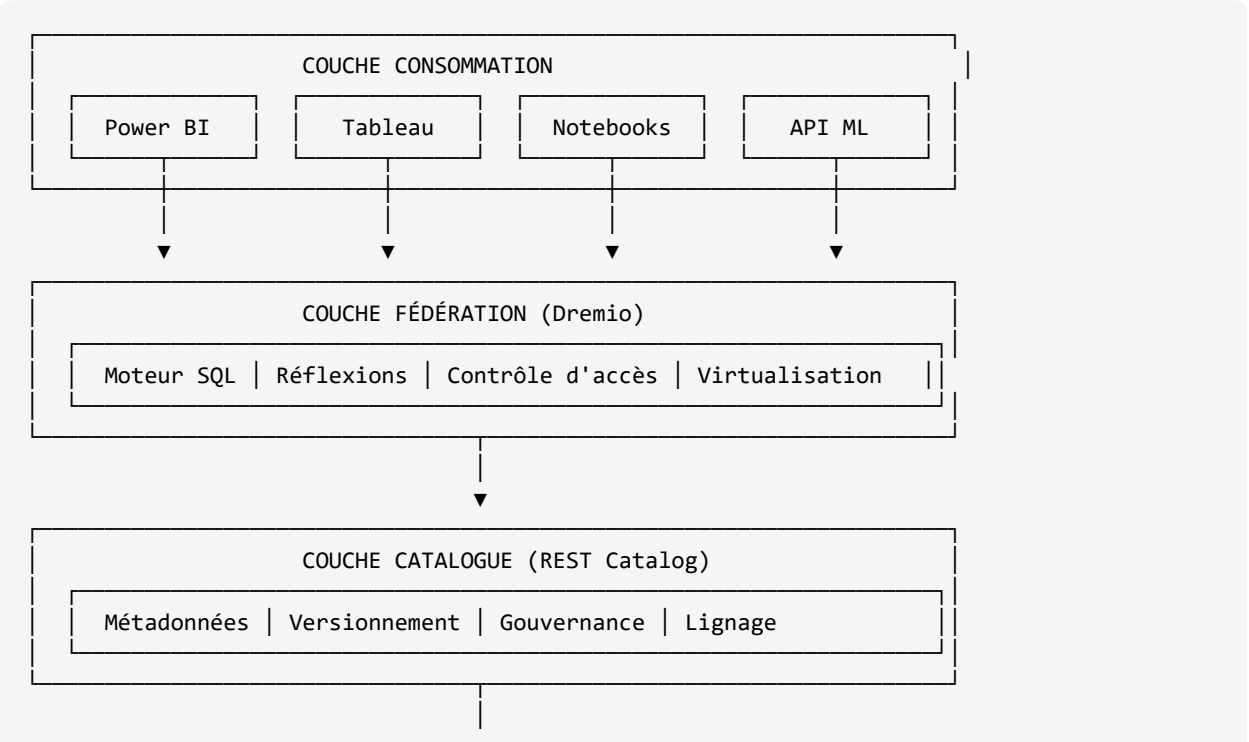
- Interface SQL unifiée pour tous les consommateurs
- Réflexions de données pour l'accélération des requêtes
- Connexion native aux outils BI existants
- Contrôle d'accès fin intégré

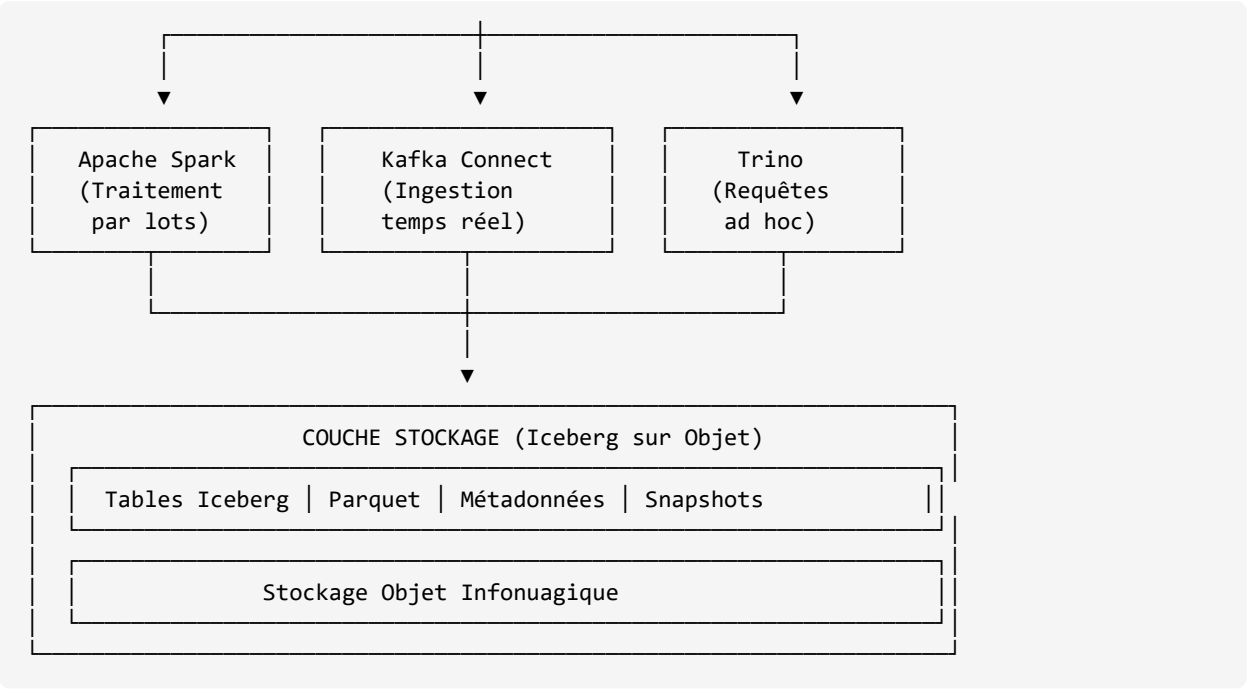
Pilier 5 : Intégration Kafka pour le temps réel

L'infrastructure Kafka existante sera étendue pour alimenter le Lakehouse en temps réel, répondant à l'exigence EF-04 (ingestion temps réel). Cette intégration exploitera les connecteurs Kafka Connect pour Iceberg, permettant l'écriture directe des flux événementiels dans les tables Iceberg. Le Volume III de cette monographie détaille les patrons d'architecture Kafka applicables.

Diagramme d'architecture cible

L'architecture cible d'Hamerliwa se structure en couches clairement définies :





Feuille de route de migration

La migration vers cette architecture cible s'étendra sur 24 mois, organisée en quatre phases principales.

Phase 1 : Fondations (Mois 1-6) - Déploiement de l'infrastructure de stockage objet - Installation et configuration du catalogue REST - Mise en place de Dremio pour la fédération - Migration pilote du domaine marketing - Formation des équipes de base

Phase 2 : Expansion (Mois 7-12) - Migration des domaines opérationnels - Intégration des flux temps réel Kafka - Déploiement des capacités de gouvernance avancées - Extension de l'adoption BI

Phase 3 : Consolidation (Mois 13-18) - Migration des domaines réglementaires - Optimisation des performances - Décommission progressive de Snowflake - Développement des cas d'usage ML

Phase 4 : Optimisation (Mois 19-24) - Décommission de l'infrastructure Hadoop - Réduction de l'empreinte Exadata - Automatisation avancée des opérations - Transfert complet aux équipes internes

Phase	Domaines	Systèmes impactés	Budget
1	Marketing	Snowflake, Hadoop	5 M$
2	Opérations	Hadoop, Azure SQL	8 M$
3	Réglementaire	Exadata (partiel), Hadoop	10 M$
4	Optimisation	Tous	5 M$

La présentation itinérante

Un plan architectural, aussi solide soit-il, ne se réalisera pas sans l'adhésion des parties prenantes. La « présentation itinérante » (*roadshow*) désigne la série de présentations adaptées à chaque audience pour construire cette adhésion. Chaque groupe de parties prenantes a des préoccupations différentes et nécessite un message personnalisé.

Présentation au Comité exécutif

L'audience : Le PDG, le CFO, le CIO et les VP des principales lignes d'affaires.

Les préoccupations : Retour sur investissement, risques stratégiques, alignement avec la vision d'entreprise.

Le message clé : Le Lakehouse Apache Iceberg permettra à Hamerliwa de développer des capacités analytiques avancées tout en réduisant les coûts à terme. C'est un investissement défensif (conformité, réduction des risques) et offensif (compétitivité, innovation).

Le contenu : - Contexte : pressions réglementaires et concurrentielles - Constats d'audit résumés en indicateurs clés - Vision architecturale en une diapositive - Analyse financière : investissement, économies, ROI sur 5 ans - Risques principaux et stratégies de mitigation - Demande : approbation du budget et du calendrier

Durée : 30 minutes + 15 minutes de questions.

Présentation au Comité de direction TI

L'audience : Le CIO, les directeurs infrastructure, applications, sécurité et données.

Les préoccupations : Faisabilité technique, impacts sur les équipes, intégration avec l'existant.

Le message clé : L'architecture proposée s'appuie sur des technologies matures et des standards ouverts. La migration progressive minimise les risques et permet un apprentissage incrémental.

Le contenu : - Architecture actuelle et ses limitations techniques - Architecture cible détaillée avec choix technologiques justifiés - Stratégie de migration par phases avec dépendances - Impacts sur les équipes et plan de formation - Métriques de succès et critères d'acceptation - Gouvernance du projet et processus de décision

Durée : 60 minutes + 30 minutes de discussion.

Présentation aux équipes métier

L'audience : Les directeurs et gestionnaires des domaines marketing, risque, finance, opérations.

Les préoccupations : Continuité des services, amélioration de leurs capacités, efforts de leur part.

Le message clé : Le nouveau Lakehouse vous donnera accès à des données plus fraîches, plus complètes et plus faciles à exploiter. Votre participation à la définition des priorités est essentielle.

Le contenu : - Frustrations actuelles validées par l'audit - Capacités nouvelles promises par le Lakehouse - Calendrier par domaine avec leurs dates clés - Rôle attendu de leurs équipes - Engagement à maintenir la continuité des services critiques - Canal de communication et de remontée des problèmes

Durée : 45 minutes avec atelier interactif.

Présentation aux équipes techniques

L'audience : Les architectes, développeurs, administrateurs et analystes de données.

Les préoccupations : Technologies à apprendre, évolution de leurs rôles, qualité de l'implémentation.

Le message clé : Le Lakehouse représente une opportunité d'acquérir des compétences recherchées sur des technologies d'avenir. Votre expertise de l'existant est précieuse pour réussir la migration.

Le contenu : - Plongée technique dans Apache Iceberg (lien avec Partie 1) - Architecture détaillée avec composants et interfaces - Stack technologique complet - Plan de formation et certifications - Processus de contribution et revue de code - Démonstration pratique de l'environnement de développement

Durée : 90 minutes avec démonstration.

> **Étude de cas : Industrielle Alliance — La communication comme facteur de succès**
> *Secteur* : Assurances et services financiers
> *Défi* : Migration d'un entrepôt de données vers une architecture Lakehouse avec résistance initiale des équipes historiques.
> *Solution* : Programme de communication structuré avec 25 sessions adaptées aux différentes audiences, complété par un site intranet dédié au projet et des bulletins hebdomadaires de progression.
> *Résultats* : Taux d'adoption de 94 % à la fin de la première phase, contre 67 % pour un projet similaire sans programme de communication.

Gestion des objections

Chaque présentation génèrera des questions et des objections légitimes. L'équipe projet doit anticiper ces objections et préparer des réponses factuelles. Une objection bien traitée renforce la crédibilité du projet ; une objection mal gérée peut compromettre l'adhésion de toute une audience.

Objection : « Pourquoi ne pas simplement étendre Snowflake ? »

Réponse : Snowflake est un excellent produit, mais crée une dépendance à un fournisseur unique avec des coûts croissants. Apache Iceberg offre la même flexibilité avec le contrôle des données et l'interopérabilité multi-fournisseur. De plus, Snowflake supporte maintenant Iceberg, donc les deux approches peuvent coexister.

Données à l'appui : Les coûts Snowflake d'Hamerliwa ont augmenté de 35 % l'an dernier pour une croissance des données de 20 %. Avec Iceberg sur stockage objet, le coût au téraoctet diminue quand le volume augmente.

Objection : « L'équipe n'a pas les compétences nécessaires »

Réponse : Le plan inclut un programme de formation substantiel et un accompagnement externe pendant la phase initiale. Les compétences Iceberg, Spark et Dremio sont transférables et valorisantes pour les carrières. Plusieurs membres de l'équipe ont déjà exprimé leur intérêt pour ces technologies.

Plan de formation : 40 heures de formation formelle par membre de l'équipe technique, complétées par un mentorat pendant 6 mois. Budget formation : 350 K$ sur la durée du projet.

Objection : « C'est trop risqué pour les systèmes réglementaires »

Réponse : C'est précisément pourquoi la migration des systèmes réglementaires est planifiée en phase 3, après que l'équipe ait acquis de l'expérience sur les domaines moins critiques. Une période de fonctionnement parallèle de 6 mois est prévue avant toute décommission.

Mécanismes de mitigation : Tests de non-régression automatisés, réconciliation quotidienne entre ancien et nouveau système, comité de validation réglementaire avant chaque décommission.

Objection : « Les coûts sont trop élevés »

Réponse : L'investissement total de 28 M$ sur 3 ans représente 1,9 fois le coût annuel actuel de la plateforme. Cependant, les économies projetées atteignent 1,2 M$/an à partir de l'année 4, générant un retour sur investissement positif en année 5. Sans cette modernisation, les coûts actuels continueront d'augmenter de 15-20 % par an.

Scénario sans action : En maintenant l'architecture actuelle, les coûts atteindront 7,2 M $/an en année 5 (projection basée sur les tendances historiques), contre 4, 1 M$/an avec le Lakehouse.

Objection : « Nous avons d'autres priorités »

Réponse : Les exigences réglementaires (Loi 25, BSIF) ne sont pas optionnelles et leurs échéances approchent. De plus, chaque mois de retard accumule de la dette technique et repousse les bénéfices. Le plan par phases permet de démarrer avec des ressources limitées et d'ajuster selon les résultats.

Calendrier réglementaire : Les nouvelles exigences BSIF sur la gestion des données entrent en vigueur en janvier 2027. La phase 3 du projet, qui adresse la conformité réglementaire, doit être complétée 3 mois avant cette date pour permettre les validations nécessaires.

Objection : « Nos concurrents utilisent une autre approche »

Réponse : Les approches varient selon les contextes. Toutefois, Apache Iceberg est devenu le standard de facto pour les nouveaux projets Lakehouse. Son adoption par Apple, Netflix, Adobe et de nombreuses institutions financières témoigne de sa maturité. En choisissant un format ouvert, Hamerliwa conserve la flexibilité d'évoluer sans réécrire toute son infrastructure.

Objection : « Le projet va perturber nos opérations quotidiennes »

Réponse : Le plan prévoit explicitement une coexistence prolongée entre l'ancien et le nouveau système. Aucune décommission ne se fera avant 6 mois de fonctionnement stable en parallèle. Les équipes opérationnelles seront impliquées dans la définition des critères de bascule et auront un droit de veto sur le calendrier de décommission.

Stratégie de gestion du changement

Au-delà des présentations formelles, la réussite du projet nécessite une stratégie de gestion du changement structurée.

Identification des champions : Dans chaque équipe impactée, identifiez un ou deux individus enthousiastes qui deviendront les ambassadeurs du projet. Ces champions reçoivent une formation avancée et participent aux décisions de conception. Leur rôle est de relayer l'information dans leurs équipes et de remonter les préoccupations.

Communication régulière : Établissez un rythme de communication prévisible. Par exemple, un bulletin hebdomadaire par courriel résumant les progrès, les prochaines étapes et les points d'attention. Un site intranet dédié au projet centralise la documentation, les questions fréquentes et les ressources de formation.

Célébration des succès : Chaque jalon atteint mérite d'être communiqué et célébré. La migration réussie du premier domaine, la première requête exécutée 10 fois plus vite qu'avant, le premier tableau de bord migré sans modification — ces succès renforcent la confiance et l'élan du projet.

Gestion des résistances : Certaines résistances sont inévitables. Plutôt que de les ignorer ou de les combattre frontalement, cherchez à comprendre leurs causes profondes. Souvent, la résistance masque une peur légitime (perte de compétences, changement de rôle, incertitude). Adressez ces peurs directement et impliquez les résistants dans la conception des solutions.

> **Étude de cas : Mouvement Desjardins — La force du réseau de champions**
> *Secteur* : Services financiers coopératifs
> *Défi* : Migration de 4 500 utilisateurs vers une nouvelle plateforme analytique dans un contexte de résistance historique au changement.
> *Solution* : Réseau de 120 champions formés intensivement, répartis dans toutes les régions et lignes d'affaires. Ces champions ont organisé plus de 300 sessions de démonstration locales et ont servi de

> premier niveau de support.
> *Résultats* : Taux d'adoption de 91 % après 6 mois, contre une cible de 75 %. Le réseau de champions a été maintenu pour les évolutions futures de la plateforme.

Documentation et gouvernance du plan

Le plan architectural et les exigences doivent être documentés formellement pour servir de référence tout au long du projet. Cette documentation inclut :

Document de vision architecturale : Synthèse exécutive de l'architecture cible, des principes directeurs et des bénéfices attendus. Ce document d'une dizaine de pages sert de référence commune à toutes les parties prenantes.

Dossier d'exigences : Liste complète des exigences fonctionnelles, non fonctionnelles, de migration et de gouvernance, avec traçabilité vers les constats d'audit et critères d'acceptation mesurables.

Plan de projet : Feuille de route détaillée avec jalons, livrables, ressources et dépendances. Ce plan sera affiné au cours des chapitres suivants qui traiteront chaque couche de l'architecture.

Registre des décisions architecturales : Journal des décisions significatives avec contexte, options considérées, décision retenue et justification. Ce registre facilite la compréhension des choix par les nouveaux membres de l'équipe et évite de revisiter des décisions déjà prises.

Matrice RACI : Clarification des rôles et responsabilités pour chaque livrable et décision majeure du projet.

IV.4.5 Résumé

Ce chapitre a établi les fondations méthodologiques pour préparer votre passage à Apache Iceberg. À travers l'étude de cas de la Banque Hamerliwa, nous avons illustré concrètement chaque étape de cette préparation critique.

Concepts clés

L'audit de plateforme constitue le point de départ indispensable de tout projet de migration. Organisé en six dimensions (architecture, données, flux, consommation, gouvernance, coûts), il fournit une compréhension factuelle de l'existant qui éclaire toutes les décisions ultérieures.

La transformation des constats en exigences traduit les observations qualitatives de l'audit en spécifications exploitables. Les exigences se structurent en quatre catégories : fonctionnelles, non fonctionnelles, de migration et de gouvernance. La matrice de traçabilité assure que chaque constat critique trouve une réponse dans les exigences.

Le plan architectural de haut niveau définit les orientations technologiques majeures avant d'entrer dans le détail de chaque composant. Pour Hamerliwa, cinq piliers structurent cette vision : Apache Iceberg comme format universel, stockage objet infonuagique, catalogue REST unifié, couche de fédération Dremio et intégration Kafka.

La présentation itinérante reconnaît que l'adhésion des parties prenantes est aussi importante que la qualité technique du plan. Chaque audience (exécutifs, direction TI, métiers, techniques) nécessite un message adapté à ses préoccupations spécifiques.

Livrables de la phase de préparation

À l'issue de cette phase, les livrables suivants doivent être disponibles :

Livrable	Contenu	Audience
Rapport d'audit	Constats par dimension, forces/faiblesses, métriques de référence	Direction, équipe projet
Cartographie des actifs	Inventaire des systèmes, flux, données	Équipe technique
Dossier d'exigences	Exigences structurées avec traçabilité	Équipe projet, fournisseurs
Document de vision	Architecture cible, principes, bénéfices	Toutes parties prenantes
Plan de projet	Phases, jalons, ressources, budget	Direction, PMO
Supports de présentation	Versions adaptées par audience	Équipe projet

Facteurs de succès

L'expérience des projets de migration similaires révèle plusieurs facteurs déterminants pour le succès :

Engagement visible de la direction : Sans un sponsor exécutif actif et visible, le projet peinera à obtenir les ressources et l'attention nécessaires. Le sponsor doit comprendre suffisamment le projet pour le défendre devant ses pairs.

Qualité de l'audit initial : Les surprises découvertes pendant l'implémentation coûtent plus cher que le temps investi dans un audit approfondi. Mieux vaut repousser le démarrage de quelques semaines pour un audit complet que de subir des retards de plusieurs mois en cours de projet.

Communication continue : La présentation itinérante n'est pas un événement ponctuel mais le début d'un processus de communication continu. Les parties prenantes doivent être informées régulièrement de la progression, des succès et des défis.

Réalisme du calendrier : Les projets de migration sous-estiment systématiquement les efforts requis. Un calendrier réaliste avec des marges pour les imprévus génère plus de confiance qu'un calendrier agressif qui sera rapidement dépassé.

Équipe dédiée : Les migrations réussies s'appuient sur une équipe dédiée au projet, pas sur des ressources partagées entre les opérations courantes et le projet. Le coût apparent de cette dédicace est compensé par l'accélération de la livraison.

Prochaines étapes

Ce chapitre conclut la préparation stratégique du passage à Apache Iceberg. Les chapitres suivants de la Partie 2 aborderont la conception détaillée de chaque couche de l'architecture :

- **Chapitre IV.5** : Sélection de la couche de stockage — évaluation des options de stockage objet et critères de décision
- **Chapitre IV.6** : Architecture de la couche d'ingestion — patrons d'ingestion par lots et temps réel, intégration Kafka

- **Chapitre IV.7** : Implémentation de la couche de catalogue — options de catalogue REST et critères de sélection
- **Chapitre IV.8** : Conception de la couche de fédération — Dremio, Trino et alternatives
- **Chapitre IV.9** : Comprendre la couche de consommation — outils BI, ML et interfaces d'accès

Chaque chapitre reprendra les exigences définies pour Hamerliwa et les utilisera pour guider les décisions architecturales spécifiques à chaque couche. Cette approche systématique assure la cohérence entre les exigences métier et les choix techniques.

Commandes et outils essentiels

Bien que ce chapitre soit principalement méthodologique, voici quelques outils utiles pour la phase d'audit :

Profilage des données avec Apache Spark :

```
# Analyse du volume et de la distribution des données
df = spark.read.parquet("s3://bucket/donnees/")
df.printSchema()
df.describe().show()
df.groupBy("partition_col").count().orderBy("count", ascending=False).show()
```

Inventaire des tables Hive/Iceberg :

```
-- Lister toutes les tables d'un namespace
SHOW TABLES IN database_name;

-- Obtenir les métadonnées d'une table
DESCRIBE EXTENDED database_name.table_name;

-- Analyser les snapshots Iceberg
SELECT * FROM database_name.table_name.snapshots;
```

Analyse des requêtes (Dremio) :

```
-- Examiner l'historique des requêtes
SELECT query_text, submitted_ts, finish_ts,
       TIMESTAMPDIFF(SECOND, submitted_ts, finish_ts) as duration_sec
FROM sys.jobs
WHERE finish_ts > CURRENT_TIMESTAMP - INTERVAL '7' DAY
ORDER BY duration_sec DESC
LIMIT 20;
```

> **Performance**
> Lors de l'audit, évitez d'exécuter des analyses lourdes pendant les heures de pointe des systèmes de production. Planifiez les requêtes de profilage pendant les fenêtres de maintenance ou utilisez des répliques de lecture si disponibles. Pour les très grands volumes, échantillonnez les données plutôt que d'analyser l'ensemble complet.

Transition vers le chapitre suivant

Avec les exigences clairement définies et le plan architectural approuvé, nous pouvons maintenant entrer dans le détail de la conception. Le Chapitre IV.5 ouvrira cette exploration en abordant la couche fondamentale de toute architecture Lakehouse : le stockage. Nous évaluerons les différentes options de stockage objet disponibles, comparerons leurs caractéristiques de performance et de coûts, et appliquerons les exigences d'Hamerliwa pour guider la sélection.

Références

- Inmon, W.H. et Linstedt, D. (2024). *Data Architecture: From Zen to Reality*. Technics Publications.
- Kimball, R. et Ross, M. (2023). *The Data Warehouse Toolkit*, 4e édition. Wiley.
- Reis, J. et Housley, M. (2022). *Fundamentals of Data Engineering*. O'Reilly Media.
- Apache Iceberg (2025). *Migration Guide*. https://iceberg.apache.org/docs/latest/migration/
- Dremio (2024). *Lakehouse Migration Best Practices*. https://www.dremio.com/resources/
- BSIF (2025). *Ligne directrice E-21 : Gestion des risques liés aux technologies de l'information*. https://www.osfi-bsif.gc.ca/
- Commission d'accès à l'information du Québec (2024). *Guide d'application de la Loi 25*. https://www.cai.gouv.qc.ca/
- Gartner (2024). *Magic Quadrant for Data Integration Tools*. Gartner Research.
- Forrester (2024). *The State of Data Management*. Forrester Research.

Chapitre IV.5 - Sélection de la Couche de Stockage

Introduction

La couche de stockage constitue le socle fondamental de toute architecture Data Lakehouse. Si Apache Iceberg fournit l'intelligence — la gestion des métadonnées, l'évolution de schéma, le partitionnement masqué —, c'est bien le stockage sous-jacent qui héberge physiquement les données et détermine les caractéristiques opérationnelles de votre plateforme. Un choix judicieux de cette couche influence directement la performance des requêtes, les coûts d'exploitation, la résilience des données et la capacité d'évolution de votre architecture.

Dans l'écosystème moderne du Lakehouse, le stockage objet s'est imposé comme le paradigme dominant. Contrairement aux systèmes de fichiers distribués traditionnels comme HDFS (Hadoop Distributed File System), le stockage objet offre une séparation nette entre le calcul et le stockage, une élasticité quasi infinie et un modèle économique aligné sur la consommation réelle. Cette évolution architecturale a transformé la manière dont les organisations conçoivent et opèrent leurs infrastructures de données.

Ce chapitre vous guide dans la sélection et la configuration de la couche de stockage optimale pour votre Lakehouse Iceberg. Nous examinerons les caractéristiques essentielles du stockage objet, comparerons les principales plateformes disponibles — tant infonuagiques que sur site —, établirons des critères de décision pragmatiques et illustrerons nos recommandations par des études de cas concrètes, notamment dans le contexte réglementaire canadien.

L'objectif n'est pas de désigner un « meilleur » choix universel, mais de vous outiller pour effectuer une sélection éclairée selon votre contexte spécifique : contraintes de résidence des données, intégrations existantes, profils de charge de travail et objectifs de coûts.

Fondamentaux du Stockage Objet pour Apache Iceberg

Anatomie du Stockage Objet

Le stockage objet repose sur un modèle fondamentalement différent des systèmes de fichiers hiérarchiques traditionnels. Chaque objet se compose de trois éléments : les données elles-mêmes, un identifiant unique (clé) et des métadonnées descriptives. Cette structure aplatie élimine la complexité des arborescences de répertoires et permet une scalabilité horizontale pratiquement illimitée.

Pour Apache Iceberg, le stockage objet joue un rôle double. D'une part, il héberge les fichiers de données — typiquement en format Parquet ou ORC — organisés selon la structure de partitionnement définie. D'autre part, il stocke l'arborescence de métadonnées Iceberg : fichiers de manifeste, listes de manifestes

et fichiers de métadonnées de table. Cette séparation logique au sein d'un même système de stockage simplifie considérablement l'architecture tout en préservant la cohérence transactionnelle.

```
bucket-lakehouse/
├── warehouse/
│   └── db_ventes/
│       └── transactions/
│           ├── data/
│           │   ├── date=2024-01-15/
│           │   │   ├── 00001-abc123.parquet
│           │   │   └── 00002-def456.parquet
│           │   └── date=2024-01-16/
│           │       └── 00001-ghi789.parquet
│           └── metadata/
│               ├── v1.metadata.json
│               ├── v2.metadata.json
│               ├── snap-123456789.avro
│               └── manifest-list-001.avro
```

Cette organisation exploite le concept de préfixes (pseudo-répertoires) pour regrouper logiquement les objets, tout en bénéficiant de la nature plate du stockage objet qui favorise les opérations parallèles à grande échelle.

Exigences Spécifiques d'Iceberg

Apache Iceberg impose certaines exigences au système de stockage sous-jacent pour garantir ses propriétés transactionnelles et sa performance.

Cohérence des lectures après écriture : Iceberg s'appuie sur la garantie que toute donnée écrite soit immédiatement lisible par les requêtes subséquentes. Historiquement, certains systèmes de stockage objet offraient uniquement une cohérence éventuelle, ce qui pouvait mener à des lectures incohérentes. Les principaux fournisseurs infonuagiques ont depuis corrigé cette limitation — Amazon S3 offre la cohérence forte depuis décembre 2020.

Support des opérations atomiques : Le mécanisme de commit d'Iceberg repose sur le remplacement atomique du pointeur de métadonnées. Cette opération doit être atomique pour éviter les conditions de course entre écrivains concurrents. Le stockage objet supporte généralement cette sémantique via des opérations conditionnelles (« conditional PUT ») ou des mécanismes de verrouillage externes.

Listage efficace des objets : Les opérations de planification de requêtes Iceberg nécessitent l'énumération des fichiers de manifeste. Un système de stockage offrant un listage rapide et paginé améliore significativement les temps de planification, particulièrement pour les tables volumineuses.

Débit soutenu pour lectures parallèles : Les moteurs de requête comme Trino ou Apache Spark lisent simultanément de nombreux fichiers de données. Le stockage doit supporter un débit agrégé élevé sans dégradation, même sous forte charge concurrente.

Séparation Calcul-Stockage : Implications Architecturales

L'architecture découplée du Lakehouse — où le stockage est indépendant des moteurs de calcul — constitue un avantage stratégique majeur. Cette séparation permet de dimensionner indépendamment les ressources de calcul (pour les requêtes) et de stockage (pour la persistance), optimisant ainsi les coûts selon les profils d'utilisation réels.

Concrètement, cette architecture signifie que vous pouvez :

- Faire évoluer la capacité de stockage sans reconfigurer les clusters de calcul
- Utiliser simultanément plusieurs moteurs de requête (Trino, Spark, Dremio) sur les mêmes données
- Suspendre complètement les ressources de calcul durant les périodes d'inactivité tout en conservant les données
- Migrer entre moteurs de calcul sans déplacer les données

Toutefois, cette séparation introduit une latence réseau entre le calcul et le stockage. L'optimisation de cette communication devient un facteur critique de performance, influençant les choix de localisation des ressources et les stratégies de mise en cache.

Panorama des Solutions de Stockage

Stockage Infonuagique Public

Amazon S3 (Simple Storage Service)

Amazon S3 demeure la référence du stockage objet, avec plus de 15 années de maturité et une adoption massive dans l'écosystème des données. Pour Apache Iceberg, S3 offre une intégration native et éprouvée.

Caractéristiques clés :

- Durabilité de 99,999999999 % (11 neuf) par réplication automatique
- Cohérence forte pour toutes les opérations depuis décembre 2020
- Classes de stockage multiples (Standard, Intelligent-Tiering, Glacier) pour optimisation des coûts
- Intégration native avec l'écosystème AWS (Athena, EMR, Glue)
- Support des points d'accès pour isolation des charges de travail

Classes de stockage pertinentes pour Iceberg :

Classe	Cas d'usage Lakehouse	Latence d'accès
S3 Standard	Données actives, requêtes fréquentes	Millisecondes
S3 Intelligent-Tiering	Données avec accès variable	Millisecondes
S3 Standard-IA	Données historiques consultées occasionnellement	Millisecondes
S3 Glacier Instant Retrieval	Archives avec besoin d'accès rapide	Millisecondes
S3 Glacier Deep Archive	Conservation réglementaire long terme	Heures

La fonctionnalité S3 Object Lock permet de satisfaire les exigences de conservation immuable (WORM — Write Once Read Many), cruciale pour la conformité réglementaire dans les secteurs financier et de la santé.

Considérations régionales canadiennes : AWS opère deux régions au Canada — Canada (Central) à Montréal et Canada (West) à Calgary. Ces régions permettent de satisfaire les exigences de résidence des données canadiennes, notamment pour les données personnelles assujetties à la LPRPDE (Loi sur la protection des renseignements personnels et les documents électroniques) et aux législations provinciales.

Azure Data Lake Storage Gen2 (ADLS Gen2)

Microsoft Azure propose ADLS Gen2 comme solution de stockage optimisée pour les charges de travail analytiques. Construit sur Azure Blob Storage, il ajoute un espace de noms hiérarchique (HNS — Hierarchical Namespace) qui améliore significativement les performances des opérations sur répertoires.

Caractéristiques clés :

- Espace de noms hiérarchique pour opérations atomiques sur répertoires
- Intégration native avec Azure Synapse Analytics et Microsoft Fabric
- Support du protocole ABFS (Azure Blob File System) optimisé pour Hadoop
- Contrôle d'accès granulaire via ACL POSIX et Azure RBAC
- Niveaux d'accès (Hot, Cool, Cold, Archive) avec politique de cycle de vie

L'espace de noms hiérarchique représente un avantage notable pour Iceberg. Les opérations de renommage de répertoires — utilisées lors des commits — deviennent atomiques et performantes, contrairement au stockage objet traditionnel où elles nécessitent une copie suivie d'une suppression.

Intégration Microsoft Fabric : ADLS Gen2 constitue le fondement d'OneLake, le lac de données unifié de Microsoft Fabric. Les organisations investies dans l'écosystème Microsoft bénéficient d'une intégration fluide entre le stockage Iceberg et les capacités analytiques de Fabric, incluant Power BI Direct Lake pour l'analytique en temps réel.

Régions canadiennes : Azure opère trois régions au Canada — Canada Central (Toronto), Canada East (Québec) et, plus récemment, Canada West (Calgary, en préversion). Cette couverture géographique facilite la conformité aux exigences de résidence des données provinciales.

Google Cloud Storage (GCS)

Google Cloud Storage offre une solution de stockage objet performante avec une intégration étroite à BigQuery et l'écosystème analytique de Google.

Caractéristiques clés :

- Classes de stockage unifiées (Standard, Nearline, Coldline, Archive)
- Cohérence forte par défaut
- Intégration native avec BigQuery et Dataproc
- Support des buckets bi-régionaux et multi-régionaux pour haute disponibilité
- Turbo Replication pour réplication accélérée entre régions

Classes de stockage :

Classe	Disponibilité minimale des objets	Coût de récupération
Standard	Aucune	Aucun
Nearline	30 jours	Modéré
Coldline	90 jours	Élevé
Archive	365 jours	Très élevé

Considérations canadiennes : Google Cloud opère deux régions au Canada — northamerica-northeast1 (Montréal) et northamerica-northeast2 (Toronto). Ces régions permettent de satisfaire les exigences de résidence des données tout en bénéficiant de l'infrastructure performante de Google.

Stockage sur Site et Hybride

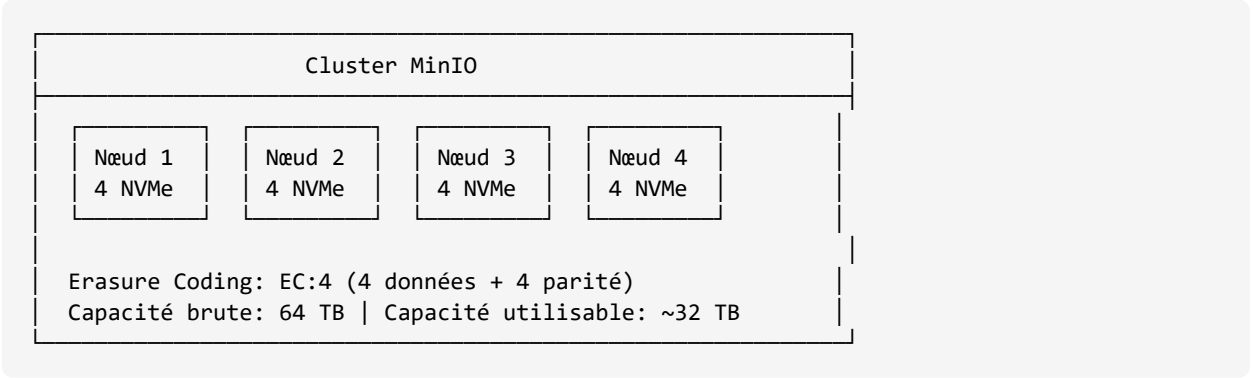
MinIO

MinIO s'est établi comme la solution de stockage objet de référence pour les déploiements sur site et hybrides. Compatible avec l'API S3, MinIO permet aux organisations de déployer une infrastructure de stockage objet dans leurs propres centres de données tout en préservant la compatibilité avec l'outillage conçu pour S3.

Caractéristiques clés :

- Compatibilité API S3 complète
- Performance native élevée (conçu pour le matériel moderne)
- Déploiement conteneurisé sur Kubernetes
- Erasure coding configurable pour équilibrer protection et efficacité
- Réplication site-à-site pour continuité des affaires

Architecture de déploiement typique :



MinIO convient particulièrement aux organisations ayant des contraintes strictes de souveraineté des données ou des investissements significatifs en infrastructure existante. Son modèle de licence permet un déploiement gratuit pour les cas d'utilisation de base, avec un support commercial disponible pour les déploiements critiques.

> **Performance**
>
> MinIO revendique des performances supérieures aux services de stockage objet infonuagiques dans des configurations comparables. Des tests internes montrent des débits dépassant 325 GiB/s en lecture et 165 GiB/s en écriture sur des grappes NVMe correctement dimensionnées. Pour les charges de travail Iceberg intensives, cette performance native peut réduire significativement les temps de requête.

Dell ECS (Elastic Cloud Storage)

Dell ECS représente une option de stockage objet de classe entreprise pour les organisations privilégiant les solutions matérielles intégrées. Conçu pour les déploiements à grande échelle, ECS offre une plate-forme de stockage géo-distribuée avec conformité S3.

Caractéristiques clés :

- Plateforme de stockage software-defined sur matériel Dell
- Réplication géographique active-active

- Support multi-protocole (S3, NFS, HDFS)
- Intégration avec les solutions de sauvegarde Dell
- Certification pour charges de travail réglementées

ECS convient aux organisations ayant des relations établies avec Dell et des exigences strictes de support matériel et logiciel intégré.

Ceph avec Interface RGW (RADOS Gateway)

Ceph, le système de stockage distribué open source, offre une interface de stockage objet compatible S3 via son composant RADOS Gateway (RGW). Cette option attire les organisations recherchant une solution entièrement open source avec un contrôle total sur l'infrastructure.

Caractéristiques clés :

- Solution open source mature et éprouvée
- Flexibilité de déploiement (bare metal, VM, conteneurs)
- Stockage unifié (bloc, fichier, objet)
- Communauté active et écosystème de partenaires

Cependant, Ceph requiert une expertise significative pour le déploiement et l'opération. Les organisations devraient évaluer soigneusement leurs capacités internes avant de s'engager dans cette voie.

Solutions Hybrides et Multi-nuage

L'architecture hybride — combinant stockage infonuagique et sur site — répond aux besoins des organisations souhaitant équilibrer agilité, coûts et contrôle. Plusieurs approches permettent d'implémenter cette stratégie avec Apache Iceberg.

Tiering automatisé : Certaines solutions permettent de déplacer automatiquement les données entre niveaux de stockage selon des politiques d'accès. Les données fréquemment consultées résident dans un stockage performant (sur site ou infonuagique premium), tandis que les données historiques migrent vers des niveaux économiques.

Réplication bidirectionnelle : Des outils comme MinIO Bucket Replication ou les fonctionnalités natives des fournisseurs infonuagiques permettent de maintenir des copies synchronisées entre environnements. Cette approche supporte les scénarios de reprise après sinistre et de migration progressive.

Abstraction multi-nuage : Des solutions comme NetApp Cloud Volumes ou Portworx fournissent une couche d'abstraction permettant de consommer le stockage de manière uniforme indépendamment du fournisseur sous-jacent. Cette approche simplifie les architectures multi-nuage au prix d'une complexité supplémentaire.

Critères de Sélection

Performance et Latence

La performance du stockage influence directement les temps de réponse des requêtes analytiques. Plusieurs métriques caractérisent cette performance dans le contexte d'un Lakehouse Iceberg.

Latence du premier octet (TTFB — Time to First Byte) : Mesure le délai entre l'initiation d'une requête de lecture et la réception du premier octet de données. Pour les requêtes analytiques lisant de nombreux petits fichiers de métadonnées, cette latence se cumule rapidement. Les services infonuagiques offrent typiquement des TTFB de 50 à 200 millisecondes, tandis que MinIO sur site peut descendre sous 10 millisecondes.

Débit soutenu : Représente le volume de données transférable par unité de temps lors de lectures ou écritures prolongées. Les moteurs de requête lisent souvent des gigaoctets de données Parquet en parallèle ; un débit insuffisant devient le goulot d'étranglement.

Opérations par seconde (IOPS) : Mesure la capacité à traiter de nombreuses requêtes concurrentes. Les opérations de listage d'objets et de lecture de métadonnées génèrent de nombreuses requêtes de faible volume.

Solution	TTFB typique	Débit lecture	IOPS pratiques
Amazon S3	50-150 ms	5+ Gbps/connexion	~3 500 GET/s/préfixe
ADLS Gen2	30-100 ms	Selon tier	Variable
GCS	50-150 ms	Élevé	Variable
MinIO (NVMe)	5-20 ms	Selon réseau	Très élevé

Stratégies d'optimisation de performance :

1. **Localisation des données** : Placez le stockage dans la même région ou zone de disponibilité que vos clusters de calcul pour minimiser la latence réseau.
2. **Partitionnement des préfixes** : S3 et d'autres services limitent les IOPS par préfixe. Distribuez vos données sur plusieurs préfixes pour paralléliser les opérations.
3. **Mise en cache des métadonnées** : Configurez le cache de métadonnées de votre moteur de requête pour éviter les lectures répétées des fichiers de manifeste.
4. **Dimensionnement des fichiers** : Ciblez des fichiers de données de 128 Mo à 1 Go pour équilibrer parallélisme et surcharge de lecture.

Coûts et Modèle Économique

Le coût total de possession (TCO — Total Cost of Ownership) du stockage comprend plusieurs composantes souvent sous-estimées lors de l'évaluation initiale.

Composantes de coût du stockage infonuagique :

Composante	Description	Impact typique
Stockage	Volume de données stockées	40-60 % du coût
Requêtes	Opérations GET, PUT, LIST	10-30 % du coût
Transfert sortant	Données sortant de la région	10-30 % du coût
Transfert entre régions	Réplication multi-région	Variable
Retrieval (archives)	Restauration depuis tiers froids	Variable

Exemple de projection de coûts mensuels (basé sur les tarifs publics de janvier 2026, région Canada Central) :

```
Scénario : Lakehouse de 100 To, 10 M requêtes/mois, 5 To sortant/mois

Amazon S3 Standard:
  Stockage: 100 To × 0,023 $/Go = 2 300 $
  Requêtes GET: 8 M × 0,0004 $/1000 = 3,20 $
  Requêtes PUT: 2 M × 0,005 $/1000 = 10,00 $
  Transfert sortant: 5 To × 0,09 $/Go = 450 $
  Total estimé: ~2 763 $/mois

Azure ADLS Gen2 (Hot tier):
  Stockage: 100 To × 0,021 $/Go = 2 100 $
  Opérations lecture: 8 M × 0,005 $/10000 = 4,00 $
  Opérations écriture: 2 M × 0,07 $/10000 = 14,00 $
  Transfert sortant: 5 To × 0,087 $/Go = 435 $
  Total estimé: ~2 553 $/mois

MinIO sur site (amorti sur 3 ans):
  Infrastructure: ~1 500 $/mois (serveurs, disques, réseau)
  Exploitation: ~500 $/mois (électricité, personnel)
  Total estimé: ~2 000 $/mois
  (Pas de coût de transfert si consommé sur site)
```

Ces estimations illustrent que le stockage infonuagique devient économiquement avantageux pour les charges de travail variables ou en croissance, tandis que le déploiement sur site peut s'avérer compétitif pour les volumes stables et élevés avec utilisation locale.

> **Performance**
>
> Les coûts de transfert sortant (« egress ») constituent souvent une surprise désagréable. Pour un Lakehouse alimentant des tableaux de bord Power BI hébergés hors de la région de stockage, ces coûts peuvent dépasser le coût de stockage lui-même. Évaluez attentivement vos flux de données et privilégiez la colocalisation calcul-stockage.

Stratégies d'optimisation des coûts :

1. **Tiering intelligent** : Utilisez S3 Intelligent-Tiering ou les équivalents Azure/GCS pour déplacer automatiquement les données rarement accédées vers des niveaux économiques.
2. **Compression et encodage** : Parquet avec compression Snappy ou ZSTD réduit significativement le volume stocké. Iceberg supporte nativement ces formats.
3. **Compaction régulière** : La compaction Iceberg réduit le nombre de fichiers, diminuant ainsi les coûts de listage et de métadonnées.
4. **Expiration des snapshots** : Configurez l'expiration automatique des snapshots anciens pour libérer l'espace des fichiers orphelins.
5. **Réservation de capacité** : AWS propose des Savings Plans et des Reserved Capacity pour le stockage à volume prévisible.

Résidence et Souveraineté des Données

Pour les organisations canadiennes, la conformité aux exigences de résidence des données représente souvent un critère déterminant dans la sélection du stockage.

Cadre réglementaire canadien :

- **LPRPDE** : La Loi sur la protection des renseignements personnels et les documents électroniques n'interdit pas explicitement le transfert transfrontalier, mais exige que les organisations assurent une protection adéquate des données.
- **Loi 25 (Québec)** : Impose des obligations accrues pour le transfert de renseignements personnels hors Québec, incluant une évaluation des facteurs de risque.
- **Secteur public fédéral** : Les politiques du Conseil du Trésor exigent généralement que les données sensibles résident au Canada.
- **Secteur financier** : Le BSIF (Bureau du surintendant des institutions financières) impose des exigences spécifiques pour l'externalisation et l'infonuagique.

Options de conformité par fournisseur :

Fournisseur	Régions canadiennes	Certifications
AWS	ca-central-1 (Montréal), ca-west-1 (Calgary)	SOC, ISO, PCI-DSS, Protected B
Azure	Canada Central, Canada East, Canada West	SOC, ISO, PCI-DSS, Protected B
GCS	northamerica-northeast1/2 (Montréal/Toronto)	SOC, ISO, PCI-DSS
MinIO	Selon déploiement	Selon certification obtenue

> **Étude de cas : Banque régionale canadienne**
>
> *Secteur* : Services financiers
>
> *Défi* : Moderniser l'infrastructure analytique tout en respectant les exigences du BSIF concernant la résidence des données au Canada
>
> *Solution* : Déploiement d'un Lakehouse Iceberg sur ADLS Gen2 dans la région Canada Central, avec chiffrement par clés gérées par le client (CMK) dans Azure Key Vault hébergé au Canada
>
> *Résultats* : Conformité maintenue, réduction de 40 % des coûts d'infrastructure analytique, temps de développement des rapports réduit de 60 %

Intégration avec l'Écosystème Existant

L'intégration harmonieuse avec votre écosystème technologique existant influence significativement la complexité d'implémentation et les coûts d'exploitation.

Matrice d'intégration :

Composant	AWS S3	ADLS Gen2	GCS	MinIO
Apache Spark	Native	Native	Native	Via S3A
Trino	Native	Native	Native	Via S3
Dremio	Native	Native	Native	Via S3
Apache Kafka	Confluent S3 Sink	Azure Blob Sink	GCS Sink	Via S3
Microsoft Fabric	Via shortcuts	Native OneLake	Via shortcuts	Non supporté
Power BI	Via Synapse	Direct Lake	Via BigQuery	Non direct
DBT	Native	Native	Native	Via S3

Considérations d'intégration clés :

1. **Authentification unifiée** : Privilégiez les solutions permettant une authentification fédérée (SAML, OIDC) pour simplifier la gestion des identités.
2. **Réseau** : Évaluez les options de connectivité privée (AWS PrivateLink, Azure Private Endpoint) pour sécuriser les communications.
3. **Surveillance** : Intégrez les métriques de stockage à votre plateforme d'observabilité existante (Datadog, Prometheus, Azure Monitor).
4. **Automatisation** : Assurez-vous que votre outillage Infrastructure-as-Code (Terraform, Pulumi) supporte le fournisseur choisi.

Configuration Optimale par Plateforme

Amazon S3 : Configuration de Référence

Une configuration S3 optimisée pour Apache Iceberg combine plusieurs fonctionnalités pour maximiser performance, sécurité et efficacité économique.

Structure de buckets recommandée :

```
# Bucket principal pour les données
aws s3api create-bucket \
  --bucket entreprise-lakehouse-data \
  --region ca-central-1 \
  --create-bucket-configuration LocationConstraint=ca-central-1

# Bucket séparé pour les métadonnées (optionnel, pour isolation)
aws s3api create-bucket \
  --bucket entreprise-lakehouse-metadata \
  --region ca-central-1 \
  --create-bucket-configuration LocationConstraint=ca-central-1
```

Configuration du versionnement et du cycle de vie :

```
{
  "Rules": [
    {
      "ID": "tiering-donnees-historiques",
      "Status": "Enabled",
      "Filter": {
        "Prefix": "warehouse/archive/"
      },
      "Transitions": [
        {
          "Days": 90,
          "StorageClass": "STANDARD_IA"
        },
        {
          "Days": 365,
          "StorageClass": "GLACIER_IR"
        }
      ]
    },
    {
      "ID": "nettoyage-fichiers-temporaires",
      "Status": "Enabled",
      "Filter": {
        "Prefix": "temp/"
      },
      "Expiration": {
        "Days": 7
      }
    },
    {
      "ID": "expiration-versions-anciennes",
      "Status": "Enabled",
      "Filter": {},
      "NoncurrentVersionExpiration": {
        "NoncurrentDays": 30
      }
    }
  ]
}
```

Configuration du chiffrement :

```
{
  "Rules": [
    {
      "ApplyServerSideEncryptionByDefault": {
        "SSEAlgorithm": "aws:kms",
        "KMSMasterKeyID": "arn:aws:kms:ca-central-1:123456789:key/abc-123"
      },
      "BucketKeyEnabled": true
    }
  ]
}
```

L'activation de « Bucket Key » réduit significativement les appels à KMS, diminuant les coûts et améliorant les performances pour les charges de travail intensives.

Points d'accès pour isolation :

Les points d'accès S3 permettent de créer des points d'entrée distincts avec des politiques d'accès spécifiques, simplifiant la gestion des permissions pour différentes équipes ou applications.

```
{
  "Name": "ap-equipe-analytique",
  "Bucket": "entreprise-lakehouse-data",
  "VpcConfiguration": {
    "VpcId": "vpc-123abc"
  },
  "Policy": {
    "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Principal": {
          "AWS": "arn:aws:iam::123456789:role/EquipeAnalytique"
        },
        "Action": ["s3:GetObject", "s3:ListBucket"],
        "Resource": [
          "arn:aws:s3:ca-central-1:123456789:accesspoint/ap-equipe-analytique",
          "arn:aws:s3:ca-central-1:123456789:accesspoint/ap-equipe-analytique/object/*"
        ]
      }
    ]
  }
}
```

Azure ADLS Gen2 : Configuration Optimale

ADLS Gen2 offre des fonctionnalités spécifiques qui optimisent les charges de travail Iceberg, notamment l'espace de noms hiérarchique.

Création du compte de stockage :

```
# Création du compte avec HNS activé
az storage account create \
  --name entrepriselakehouse \
  --resource-group rg-data-platform \
  --location canadacentral \
  --sku Standard_LRS \
  --kind StorageV2 \
  --enable-hierarchical-namespace true \
  --min-tls-version TLS1_2

# Création du conteneur principal
az storage fs create \
  --name lakehouse \
  --account-name entrepriselakehouse
```

Structure de répertoires recommandée :

```
lakehouse/
├── bronze/           # Données brutes
│   ├── sources/
│   └── staging/
```

```
├── silver/           # Données transformées
│   └── domaines/
├── gold/             # Données agrégées
│   └── marts/
└── _metadata/        # Métadonnées Iceberg centralisées
```

Configuration des niveaux d'accès :

```
# Politique de gestion du cycle de vie
az storage account management-policy create \
  --account-name entrepriselakehouse \
  --resource-group rg-data-platform \
  --policy '{
    "rules": [
      {
        "name": "archive-donnees-historiques",
        "enabled": true,
        "type": "Lifecycle",
        "definition": {
          "filters": {
            "prefixMatch": ["lakehouse/archive/"]
          },
          "actions": {
            "baseBlob": {
              "tierToCool": {"daysAfterModificationGreaterThan": 30},
              "tierToCold": {"daysAfterModificationGreaterThan": 90},
              "tierToArchive": {"daysAfterModificationGreaterThan": 365}
            }
          }
        }
      }
    ]
  }'
```

Intégration avec Microsoft Fabric via OneLake :

Pour les organisations utilisant Microsoft Fabric, ADLS Gen2 peut être exposé comme source de données OneLake via des raccourcis (shortcuts), permettant l'accès aux tables Iceberg directement depuis Power BI en mode Direct Lake.

```
# Structure OneLake avec raccourcis vers ADLS Gen2
MonEspaceTravail/
└── MonLakehouse.Lakehouse/
    └── Tables/
        └── ventes (raccourci vers adls://entrepriselakehouse/lakehouse/gold/ventes)
```

MinIO : Déploiement de Production

Un déploiement MinIO pour un Lakehouse de production nécessite une planification soigneuse de l'infrastructure et de la configuration.

Architecture matérielle recommandée (cluster 4 nœuds) :

Compo-sant	Spécification	Justification
CPU	2× 16 cœurs (Xeon ou EPYC)	Parallélisme des opérations
RAM	128 Go	Cache et opérations d'erasure coding
Stockage	8× NVMe 4 To	Performance et capacité
Réseau	2× 25 Gbps	Débit inter-nœuds

Déploiement via Helm sur Kubernetes :

```
# values.yaml pour MinIO Operator
tenant:
  name: lakehouse-minio
  pools:
    - servers: 4
      volumesPerServer: 8
      size: 4Ti
      storageClassName: local-nvme
  requestAutoCert: true
  s3:
    bucketDNS: true
  features:
    bucketDNS: true
    domains:
      minio:
        - minio.lakehouse.internal
```

Configuration du bucket Iceberg :

```
# Création du bucket avec versionnement
mc mb lakehouse/warehouse --with-versioning

# Configuration du cycle de vie
mc ilm add lakehouse/warehouse \
  --transition-days 90 \
  --storage-class WARM \
  --prefix "archive/"

# Configuration de la réplication (pour DR)
mc admin bucket remote add lakehouse/warehouse \
  https://minio-dr.site-secondaire.internal/warehouse \
  --service replication
```

Optimisation des performances :

```
# Augmentation des connexions parallèles
mc admin config set lakehouse api requests_max=10000

# Activation du cache de métadonnées
mc admin config set lakehouse cache drives=/cache-nvme \
  expiry=90 \
  quota=80
```

Stratégies de Migration

Migration depuis HDFS

La migration depuis HDFS vers le stockage objet représente un scénario fréquent pour les organisations modernisant leur infrastructure Hadoop. Apache Iceberg facilite cette transition grâce à sa capacité à référencer des données existantes sans les déplacer immédiatement.

Approche progressive en trois phases :

Phase 1 : Migration des métadonnées Créez des tables Iceberg pointant vers les données HDFS existantes. Cette étape ne déplace pas les données mais établit la couche de métadonnées Iceberg.

```
-- Création d'une table Iceberg référençant des données HDFS existantes
CALL system.register_table(
  table => 'lakehouse.ventes_historiques',
  metadata_file => 'hdfs://cluster/warehouse/ventes/metadata/v1.metadata.json'
);
```

Phase 2 : Migration incrémentale des données Copiez progressivement les données vers le stockage objet, en commençant par les partitions les plus anciennes (moins fréquemment accédées).

```
-- Réécriture d'une partition vers S3
CALL lakehouse.system.rewrite_data_files(
  table => 'lakehouse.ventes_historiques',
  where => 'date_partition < date ''2023-01-01''',
  options => map('target-file-size-bytes', '536870912')
);
```

Phase 3 : Basculement complet Une fois toutes les données migrées, mettez à jour les chemins de stockage par défaut et décommissionnez HDFS.

> **Migration**
>
> *De* : Cluster HDFS avec tables Hive
>
> *Vers* : Lakehouse Iceberg sur S3
>
> *Stratégie* : Migration incrémentale avec coexistence temporaire
>
> *Durée typique* : 3-6 mois selon le volume de données
>
> *Risque* : Faible si exécutée progressivement avec validation

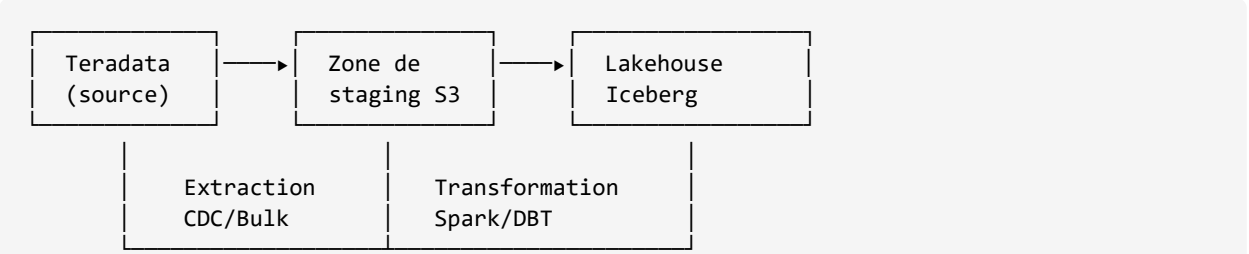
Outils de migration :

- **DistCp** : Utilitaire Hadoop pour copie distribuée, performant pour les grands volumes
- **AWS DataSync** : Service géré pour transfert HDFS vers S3
- **Apache NiFi** : Orchestration flexible des flux de données
- **Dremio Data Migration** : Outil commercial avec optimisations Iceberg

Migration depuis un Data Warehouse

La migration depuis un entrepôt de données traditionnel (Teradata, Oracle, SQL Server) vers un Lakehouse Iceberg requiert une approche différente, axée sur l'extraction et la transformation des données.

Architecture de migration type :



Considérations clés :

1. **Extraction** : Utilisez des outils CDC (Change Data Capture) pour synchroniser les données pendant la période de transition
2. **Schéma** : Traduisez les types de données et contraintes vers les équivalents Iceberg
3. **Performance** : Optimisez le partitionnement Iceberg selon les patterns de requête, pas selon le partitionnement source
4. **Validation** : Implémentez des contrôles de qualité comparant source et destination

> **Étude de cas : Détaillant canadien**
>
> *Secteur* : Commerce de détail
>
> *Défi* : Migrer 15 To de données analytiques depuis Teradata vers un Lakehouse économique tout en maintenant les SLA de reporting
>
> *Solution* : Migration progressive vers Iceberg sur Azure ADLS Gen2, avec phase de coexistence de 4 mois. DBT pour transformation, Dremio pour requêtes fédérées pendant la transition
>
> *Résultats* : Réduction de 70 % des coûts d'infrastructure, flexibilité accrue pour intégrer de nouvelles sources de données, performance des requêtes maintenue grâce au partitionnement masqué

Migration Multi-nuage

Les scénarios de migration entre fournisseurs infonuagiques présentent des défis spécifiques liés aux coûts de transfert et aux différences d'API.

Stratégie de migration inter-nuage :

1. **Évaluation des coûts de transfert** : Les coûts d'egress peuvent atteindre 0,09 $/Go. Pour 100 To, prévoyez ~9 000 $ en coûts de transfert.
2. **Transfert direct vs intermédiaire** : Le transfert direct entre fournisseurs est généralement plus économique que via un intermédiaire sur site.
3. **Outils de transfert** :
   - **Rclone** : Outil open source supportant tous les fournisseurs majeurs
   - **Services gérés** : AWS DataSync, Azure Data Factory, Google Transfer Service
4. **Validation de l'intégrité** : Vérifiez les checksums MD5 ou SHA256 après transfert

```
# Exemple de transfert S3 vers GCS avec rclone
rclone copy s3:lakehouse-source gcs:lakehouse-destination \
  --transfers 32 \
  --checkers 16 \
  --checksum \
  --progress
```

Mise à jour des chemins Iceberg :

Après migration du stockage, mettez à jour les références dans les métadonnées Iceberg :

```
-- Enregistrement de la table avec le nouveau chemin
CALL system.register_table(
  table => 'nouveau_catalog.nouvelle_db.ma_table',
  metadata_file => 'gs://lakehouse-destination/warehouse/ma_table/metadata/v42.metadata.
json'
);
```

Haute Disponibilité et Reprise après Sinistre

Architectures de Résilience

La conception d'une architecture résiliente pour le Lakehouse nécessite de considérer plusieurs niveaux de protection.

Niveau 1 : Durabilité du stockage Les fournisseurs infonuagiques offrent une durabilité intrinsèque via la réplication au sein d'une région. S3 Standard, par exemple, réplique les données sur au minimum trois zones de disponibilité.

Niveau 2 : Réplication inter-régionale Pour la protection contre les sinistres régionaux, configurez la réplication vers une région secondaire.

```
// Configuration de réplication S3
{
  "Role": "arn:aws:iam::123456789:role/s3-replication-role",
  "Rules": [
    {
      "Status": "Enabled",
      "Priority": 1,
      "Filter": {
        "Prefix": "warehouse/"
      },
      "Destination": {
        "Bucket": "arn:aws:s3:::lakehouse-dr-west",
        "ReplicationTime": {
          "Status": "Enabled",
          "Time": {"Minutes": 15}
        },
        "Metrics": {
          "Status": "Enabled",
          "EventThreshold": {"Minutes": 15}
```

```
        }
      },
      "DeleteMarkerReplication": {"Status": "Enabled"}
    }
  ]
}
```

Niveau 3 : Sauvegarde des métadonnées Les métadonnées Iceberg (fichiers JSON et Avro) sont critiques pour l'accès aux données. Implémentez une sauvegarde spécifique de ces fichiers avec une rétention adaptée.

```
# Sauvegarde quotidienne des métadonnées vers un bucket dédié
aws s3 sync s3://lakehouse/warehouse/db/table/metadata/ \
  s3://lakehouse-backup/metadata/$(date +%Y-%m-%d)/db/table/ \
  --storage-class GLACIER_IR
```

Stratégies de Basculement

Le basculement vers le site de reprise après sinistre doit être planifié et testé régulièrement.

Procédure de basculement :

1. **Détection** : Surveillance proactive détectant l'indisponibilité du site principal
2. **Décision** : Évaluation de la durée estimée de l'interruption vs RPO/RTO
3. **Synchronisation finale** : Si possible, synchronisation des dernières données
4. **Basculement du catalogue** : Mise à jour des références vers le stockage DR
5. **Validation** : Vérification de l'accessibilité et de l'intégrité des données
6. **Communication** : Notification des équipes et ajustement des configurations clientes

RPO et RTO typiques :

Configuration	RPO	RTO	Coût relatif
Région unique, multi-AZ	0	Minutes	$
Réplication asynchrone inter-région	Minutes à heures	Heures	$
$\|\|$ $Réplicationasynchroneinter-région$ $\|$ 0 $\|$ $Minu$			
$\|\|$ $Multi-nuageactif-passif$ $\|$ $Minutes$ $\|$ $Heur$			
			$

Sauvegarde et Restauration Iceberg

Apache Iceberg offre des mécanismes natifs facilitant la sauvegarde et la restauration.

Time Travel pour restauration : Le Time Travel Iceberg permet de restaurer l'état d'une table à un moment antérieur sans restauration physique des données.

```
-- Restauration vers un snapshot spécifique
CALL lakehouse.system.rollback_to_snapshot('db.table', 1234567890);
```

```
-- Ou vers un timestamp
CALL lakehouse.system.rollback_to_timestamp('db.table', TIMESTAMP '2024-01-15 10:00:00');
```

Sauvegarde complète via export :

```
-- Export d'une table vers un emplacement de sauvegarde
CREATE TABLE lakehouse_backup.db.table_backup
WITH (
  format = 'PARQUET',
  location = 's3://backup-bucket/db/table/'
)
AS SELECT * FROM lakehouse.db.table;
```

Restauration depuis sauvegarde :

```
-- Importation depuis sauvegarde
CALL system.register_table(
  table => 'lakehouse.db.table_restored',
  metadata_file => 's3://backup-bucket/metadata/db/table/v42.metadata.json'
);
```

Optimisation des Performances de Stockage

Dimensionnement des Fichiers

La taille des fichiers de données influence directement les performances des requêtes. Des fichiers trop petits multiplient les opérations de métadonnées, tandis que des fichiers trop volumineux limitent le parallélisme.

Recommandations de dimensionnement :

Type de charge de travail	Taille cible	Justification
Requêtes analytiques standard	256 Mo - 512 Mo	Équilibre parallélisme/surcharge
Streaming micro-batch	64 Mo - 128 Mo	Latence réduite
Archivage long terme	512 Mo - 1 Go	Efficacité de stockage
Tables de référence	32 Mo - 64 Mo	Lectures fréquentes complètes

Configuration Iceberg pour dimensionnement :

```
-- Définition de la taille cible lors de la création
CREATE TABLE lakehouse.db.ventes (
  id BIGINT,
  montant DECIMAL(10,2),
  date_transaction DATE
)
```

```
WITH (
  'write.target-file-size-bytes' = '268435456',  -- 256 Mo
  'write.distribution-mode' = 'hash'
);

-- Ajustement via ALTER TABLE
ALTER TABLE lakehouse.db.ventes
SET PROPERTIES (
  'write.target-file-size-bytes' = '536870912'  -- 512 Mo
);
```

Stratégies de Compaction

La compaction consolide les petits fichiers résultant d'écritures incrémentales en fichiers plus volumineux, améliorant les performances de lecture.

Types de compaction :

1. **Compaction par fusion (bin-packing)** : Combine les fichiers sous-dimensionnés
2. **Compaction avec tri (sort)** : Réorganise les données selon un ordre optimal
3. **Compaction Z-order** : Optimise pour les requêtes multi-colonnes

```
-- Compaction standard
CALL lakehouse.system.rewrite_data_files(
  table => 'lakehouse.db.ventes',
  options => map(
    'target-file-size-bytes', '536870912',
    'min-input-files', '5'
  )
);

-- Compaction avec tri
CALL lakehouse.system.rewrite_data_files(
  table => 'lakehouse.db.ventes',
  strategy => 'sort',
  sort_order => 'date_transaction ASC NULLS LAST, region'
);
```

> **Performance**
>
> La compaction Z-order peut améliorer les performances de requêtes multi-colonnes de 2× à 10× pour les tables volumineuses. Cependant, elle est coûteuse en ressources de calcul. Planifiez-la durant les périodes de faible activité.

Mise en Cache et Accélération

Plusieurs niveaux de cache peuvent accélérer les accès au stockage objet.

Cache de métadonnées : Les moteurs de requête comme Trino et Spark maintiennent un cache des métadonnées Iceberg pour éviter les lectures répétées.

```
# Configuration Trino pour cache de métadonnées
iceberg.metadata.cache-ttl=5m
iceberg.metadata.cache-size=1000
```

Cache de données local : Des solutions comme Alluxio ou le cache natif de Dremio accélèrent les lectures répétées en conservant les données fréquemment accédées sur stockage local rapide.

```
# Configuration Alluxio comme cache pour S3
alluxio.master.mount.table.root.ufs=s3://lakehouse/warehouse
alluxio.user.file.metadata.sync.interval=1m
alluxio.user.block.size.bytes.default=256MB
```

Accélérateurs infonuagiques :

- **S3 Express One Zone** : Classe de stockage à faible latence pour données chaudes
- **Azure Premium Blob Storage** : Performance SSD pour charges critiques
- **GCS Turbo Replication** : Accélère la disponibilité des données répliquées

---

Études de Cas Canadiennes

Secteur Financier : Coopérative de Services Financiers

> **Étude de cas : Grande coopérative financière québécoise**
>
> *Secteur* : Services financiers coopératifs
>
> *Défi* : Consolider 25 silos de données analytiques tout en respectant les exigences du BSIF et de l'AMF concernant la résidence des données au Canada. L'infrastructure existante Teradata atteignait ses limites de capacité avec des coûts croissants.
>
> *Solution* : Architecture Lakehouse Iceberg sur ADLS Gen2 dans les régions Canada Central et Canada East, avec réplication synchrone pour haute disponibilité. Chiffrement par clés gérées par le client (CMK) stockées dans Azure Key Vault canadien. Intégration avec Microsoft Fabric pour l'analytique libre-service.
>
> *Résultats* :
>
> - Réduction de 55 % du TCO sur 3 ans
> - Consolidation de 25 silos en plateforme unifiée
> - Temps de préparation des données réduit de 80 %
> - Conformité maintenue avec audit réussi du BSIF

Cette coopérative a choisi ADLS Gen2 principalement pour son intégration native avec Microsoft Fabric et Power BI, outils déjà répandus dans l'organisation. L'espace de noms hiérarchique a simplifié la migration depuis leur structure de fichiers existante.

Secteur Énergie : Société d'État Hydroélectrique

Étude de cas : Producteur d'hydroélectricité majeur

Secteur : Énergie et services publics

Défi : Gérer et analyser 50 To de données de télémétrie quotidiennes provenant de milliers de capteurs IoT répartis sur le réseau de production et distribution. Les systèmes existants ne pouvaient pas absorber ce volume tout en permettant des analyses en temps quasi réel.

Solution : Streaming Lakehouse combinant Apache Kafka (tel que documenté dans le Volume III) et Apache Iceberg sur Amazon S3 dans la région ca-central-1. Architecture Lambda avec couche streaming pour alertes temps réel et couche batch pour analyses historiques.

Résultats :

- Ingestion de 50 To/jour avec latence < 5 minutes
- Détection précoce d'anomalies réduisant les pannes de 30 %
- Capacité d'analyse historique sur 10 ans de données
- Coûts de stockage optimisés via tiering automatique (70 % des données en Glacier)

Le choix de S3 a été motivé par la maturité de l'écosystème Kafka-S3 (connecteurs Confluent) et l'élasticité nécessaire pour absorber les pics de télémétrie lors d'événements météorologiques.

Secteur Santé : Réseau de Recherche Clinique

Étude de cas : Réseau pan-canadien de recherche en santé

Secteur : Santé et recherche biomédicale

Défi : Fédérer les données de recherche clinique de 15 établissements à travers le Canada tout en respectant les réglementations provinciales variées sur les données de santé. Chaque province impose des exigences différentes de résidence et de consentement.

Solution : Architecture hybride avec MinIO déployé dans chaque établissement pour le stockage primaire, et réplication sélective vers GCS (région Montréal) pour les analyses pan-canadiennes. Le catalogue Iceberg centralise les métadonnées tandis que les données demeurent dans leur juridiction d'origine.

Résultats :

- Conformité avec les réglementations de 10 provinces
- Réduction de 90 % du temps de préparation des cohortes de recherche
- Capacité d'analyse fédérée sans déplacement des données sensibles
- Économies de 40 % par rapport à une solution commerciale centralisée

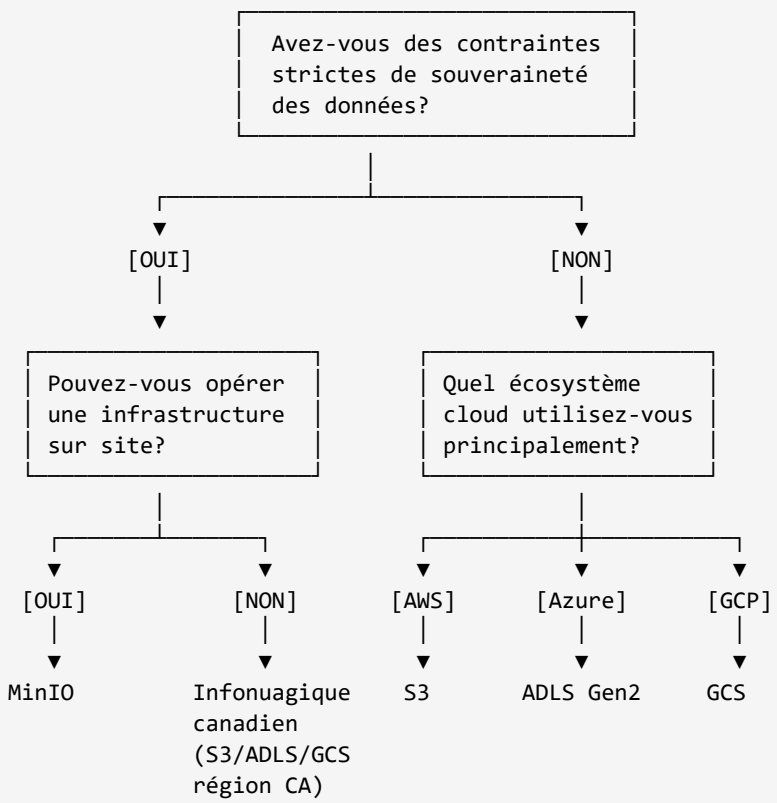
Cette architecture illustre la flexibilité d'Iceberg pour les scénarios de données distribuées. Le catalogue REST centralise la découverte des données tandis que le stockage physique respecte les contraintes juridictionnelles.

Matrice de Décision

Tableau Comparatif des Solutions

Critère	AWS S3	ADLS Gen2	GCS	MinIO
Maturité Iceberg	★★★★★	★★★★☆	★★★★☆	★★★★☆
Performance	★★★★☆	★★★★☆	★★★★☆	★★★★★
Coût (volume élevé)	★★★☆☆	★★★☆☆	★★★☆☆	★★★★★
Intégration Fabric	★★★☆☆	★★★★★	★★☆☆☆	★☆☆☆☆
Régions canadiennes	★★★★☆	★★★★★	★★★☆☆	★★★★★
Complexité opérationnelle	★☆☆☆☆	★☆☆☆☆	★☆☆☆☆	★★★★☆
Flexibilité multi-nuage	★★★☆☆	★★☆☆☆	★★☆☆☆	★★★★★

Arbre de Décision



Recommandations par Profil

Startup ou PME sans investissement cloud existant : Commencez avec S3 ou ADLS Gen2 dans une région canadienne. Les deux offrent une courbe d'apprentissage faible et une tarification à l'usage sans

engagement. Choisissez ADLS Gen2 si vous anticipez une utilisation intensive de Power BI ou Microsoft Fabric.

Grande entreprise avec écosystème Microsoft : ADLS Gen2 avec Microsoft Fabric offre l'intégration la plus fluide. OneLake simplifie la gouvernance et Direct Lake optimise les performances analytiques. Les régions canadiennes satisfont les exigences réglementaires.

Organisation avec exigences strictes de souveraineté : MinIO déployé dans vos propres centres de données offre le contrôle maximal. Prévoyez une équipe d'exploitation qualifiée et des processus de sauvegarde robustes.

Architecture multi-nuage ou hybride : MinIO comme couche d'abstraction permet une portabilité maximale. Alternativement, utilisez des catalogues REST Iceberg pour fédérer des stockages hétérogènes.

Charges de travail streaming intensives : S3 avec l'écosystème Confluent offre l'intégration la plus mature pour les architectures Kafka-Iceberg. Les connecteurs Confluent S3 Sink sont éprouvés en production à grande échelle.

Conclusion

La sélection de la couche de stockage pour votre Lakehouse Apache Iceberg constitue une décision architecturale structurante dont les implications perdurent bien au-delà de l'implémentation initiale. Ce choix influence la performance opérationnelle, les coûts d'exploitation, la résilience des données et la capacité d'évolution de votre plateforme analytique.

Le stockage objet s'est imposé comme le paradigme dominant pour les architectures Lakehouse modernes, offrant la séparation calcul-stockage, l'élasticité et le modèle économique aligné sur les besoins actuels. Parmi les options disponibles, aucune ne s'impose universellement : AWS S3 excelle par sa maturité et son écosystème, ADLS Gen2 par son intégration Microsoft Fabric, GCS par sa cohérence avec l'écosystème Google, et MinIO par sa flexibilité pour les déploiements sur site.

Pour les organisations canadiennes, les considérations de résidence des données ajoutent une dimension réglementaire à cette décision. Les trois grands fournisseurs infonuagiques opèrent désormais des régions au Canada, permettant de concilier agilité infonuagique et conformité réglementaire. MinIO demeure pertinent pour les contextes exigeant un contrôle total sur l'infrastructure.

L'approche recommandée consiste à :

1. Inventorier vos contraintes non négociables (réglementaires, intégrations existantes)
2. Évaluer les profils de charge de travail (volume, fréquence d'accès, patterns de requête)
3. Projeter les coûts sur 3 à 5 ans incluant les scénarios de croissance
4. Prototyper avec des données représentatives avant l'engagement
5. Planifier les stratégies de migration et de reprise après sinistre dès la conception

Le chapitre suivant aborde l'ingestion de données dans votre Lakehouse, où nous examinerons comment alimenter efficacement cette couche de stockage que vous venez de sélectionner, en explorant les patterns d'ingestion batch et streaming.

Résumé

Concepts clés :

- Le stockage objet constitue le fondement du Lakehouse moderne, offrant durabilité, élasticité et séparation calcul-stockage
- Apache Iceberg impose des exigences spécifiques : cohérence forte, opérations atomiques, listage efficace et débit soutenu
- Les principaux fournisseurs (S3, ADLS Gen2, GCS) offrent des régions canadiennes satisfaisant les exigences de résidence des données
- MinIO permet les déploiements sur site avec compatibilité S3

Critères de sélection :

- Performance : latence, débit, IOPS selon les profils de charge
- Coûts : stockage, requêtes, transfert sortant, retrieval
- Conformité : résidence des données, certifications sectorielles
- Intégration : écosystème existant, outils analytiques

Optimisations :

- Dimensionnement des fichiers : 256-512 Mo pour charges analytiques standard
- Compaction régulière pour consolidation des petits fichiers
- Tiering automatique pour optimisation des coûts long terme
- Cache de métadonnées pour accélération des planifications de requête

Résilience :

- Réplication multi-AZ incluse dans les services infonuagiques
- Réplication inter-région pour protection contre sinistres régionaux
- Time Travel Iceberg pour restauration logique rapide
- Sauvegarde des métadonnées comme filet de sécurité

Recommandations par contexte :

- Écosystème Microsoft ⟶ ADLS Gen2 avec Fabric
- Charges streaming ⟶ S3 avec écosystème Confluent
- Souveraineté stricte ⟶ MinIO sur site
- Multi-nuage ⟶ MinIO ou catalogue REST fédéré

Ce chapitre établit les fondations de stockage de votre architecture Lakehouse. Le chapitre suivant, « Ingestion de Données dans le Lakehouse », détaille les stratégies pour alimenter efficacement cette infrastructure avec vos sources de données batch et streaming.

Chapitre IV.6 - Architecture de la Couche d'Ingestion

Introduction

La couche d'ingestion constitue l'interface vitale entre vos systèmes sources et votre Data Lakehouse. Elle détermine la fraîcheur des données disponibles pour l'analyse, la fiabilité des pipelines de données et, ultimement, la capacité de votre organisation à prendre des décisions éclairées en temps opportun. Une architecture d'ingestion bien conçue transforme le chaos des données brutes provenant de dizaines — voire de centaines — de systèmes hétérogènes en un flux ordonné alimentant vos tables Apache Iceberg.

Dans l'écosystème du Lakehouse moderne, l'ingestion ne se limite plus au traditionnel chargement batch nocturne. Les organisations exigent désormais une palette complète de latences : du temps réel pour les tableaux de bord opérationnels, au micro-batch pour les analyses tactiques, jusqu'au batch classique pour les consolidations historiques. Apache Iceberg, grâce à ses capacités transactionnelles ACID et son support natif des écritures concurrentes, permet d'implémenter cette diversité de patterns au sein d'une architecture unifiée.

Ce chapitre vous guide dans la conception et l'implémentation d'une couche d'ingestion robuste pour votre Lakehouse Iceberg. Nous examinerons les patterns fondamentaux d'ingestion, comparerons les technologies disponibles — Apache Spark, Apache Flink, Kafka Connect et leurs alternatives —, détaillerons les stratégies de chargement adaptées à chaque cas d'usage et illustrerons nos recommandations par des architectures de référence et des études de cas concrètes.

L'intégration avec Apache Kafka, documentée en profondeur dans le Volume III de cette monographie, occupe une place centrale dans notre exploration. Le Streaming Lakehouse — fusion du traitement événementiel temps réel et de l'analytique sur données historiques — représente l'aboutissement architectural vers lequel convergent les organisations les plus matures en matière de données.

Patterns Fondamentaux d'Ingestion

Taxonomie des Modes d'Ingestion

L'ingestion de données dans un Lakehouse Iceberg s'articule autour de trois patterns fondamentaux, chacun répondant à des exigences distinctes de latence, de volume et de complexité opérationnelle.

Ingestion Batch : Le pattern historique demeure pertinent pour les scénarios où la latence de plusieurs heures est acceptable. Les chargements batch traitent des volumes massifs de données en une seule opération, optimisant l'utilisation des ressources et simplifiant la gestion des erreurs. Ce mode convient aux extractions quotidiennes de systèmes ERP, aux consolidations mensuelles et aux migrations de données historiques.

Ingestion Micro-Batch : Compromis entre latence et efficacité, le micro-batch traite les données en petits lots à intervalles réguliers — typiquement toutes les 5 à 15 minutes. Ce pattern offre une fraîcheur suffisante pour la plupart des cas d'usage analytiques tout en préservant les optimisations batch comme le dimensionnement des fichiers et la compaction efficace.

Ingestion Streaming : Le traitement événement par événement — ou en micro-lots de quelques secondes — répond aux exigences de temps réel. Les tableaux de bord opérationnels, la détection de fraude et les systèmes de recommandation en temps réel nécessitent cette latence minimale. Iceberg supporte désormais nativement ce mode grâce à ses commits incrémentaux et sa gestion des conflits.

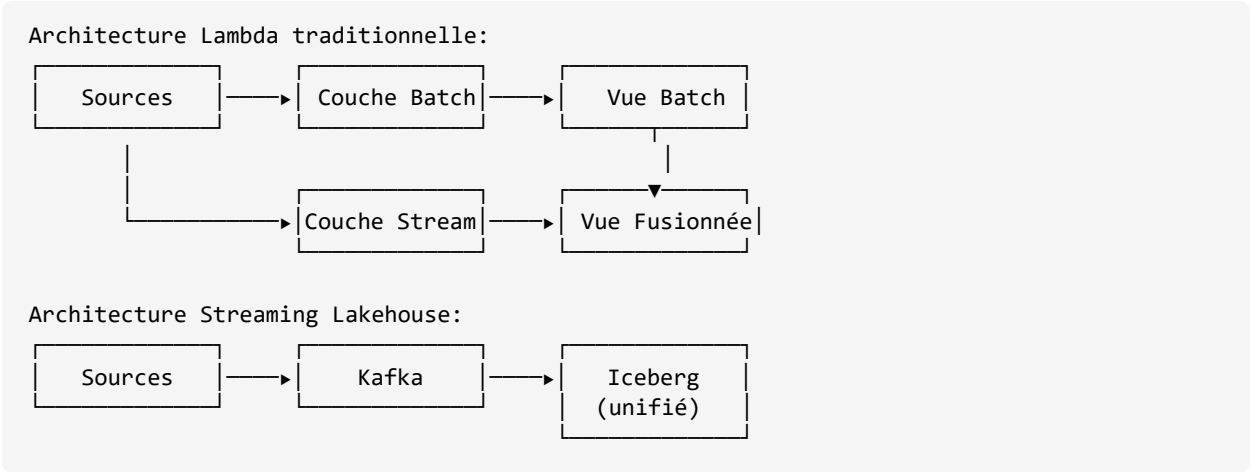
Pattern	Latence typique	Volume par opération	Complexité	Cas d'usage
Batch	Heures	Go à To	Faible	ETL nocturne, migrations
Micro-batch	Minutes	Mo à Go	Moyenne	Rapports tactiques
Streaming	Secondes	Ko à Mo	Élevée	Temps réel, alertes

Évolution vers le Streaming Lakehouse

L'architecture Streaming Lakehouse, introduite conceptuellement dans le Volume III avec Apache Kafka, unifie les traitements batch et streaming au sein d'une plateforme cohérente. Apache Iceberg joue un rôle central dans cette unification en fournissant :

- **Commits atomiques** permettant des écritures concurrentes depuis plusieurs pipelines
- **Snapshots immuables** garantissant des lectures cohérentes pendant les écritures
- **Time Travel** offrant un accès aux états historiques sans duplication
- **Schema Evolution** accommodant les changements de structure sans interruption

Cette convergence élimine la traditionnelle « architecture Lambda » avec ses chemins batch et streaming séparés, réduisant la complexité opérationnelle et les risques d'incohérence entre couches.



Sémantiques de Livraison

La fiabilité de l'ingestion repose sur la sémantique de livraison garantie par votre architecture. Trois niveaux de garantie existent :

At-most-once (au plus une fois) : Les données peuvent être perdues mais ne seront jamais dupliquées. Acceptable uniquement pour les métriques non critiques où une perte occasionnelle est tolérable.

At-least-once (au moins une fois) : Les données ne seront jamais perdues mais peuvent être dupliquées. Nécessite une logique de déduplication en aval ou des opérations idempotentes.

Exactly-once (exactement une fois) : Chaque enregistrement est traité exactement une fois, sans perte ni duplication. Cette sémantique exige une coordination entre le système source, le pipeline de traitement et le système cible.

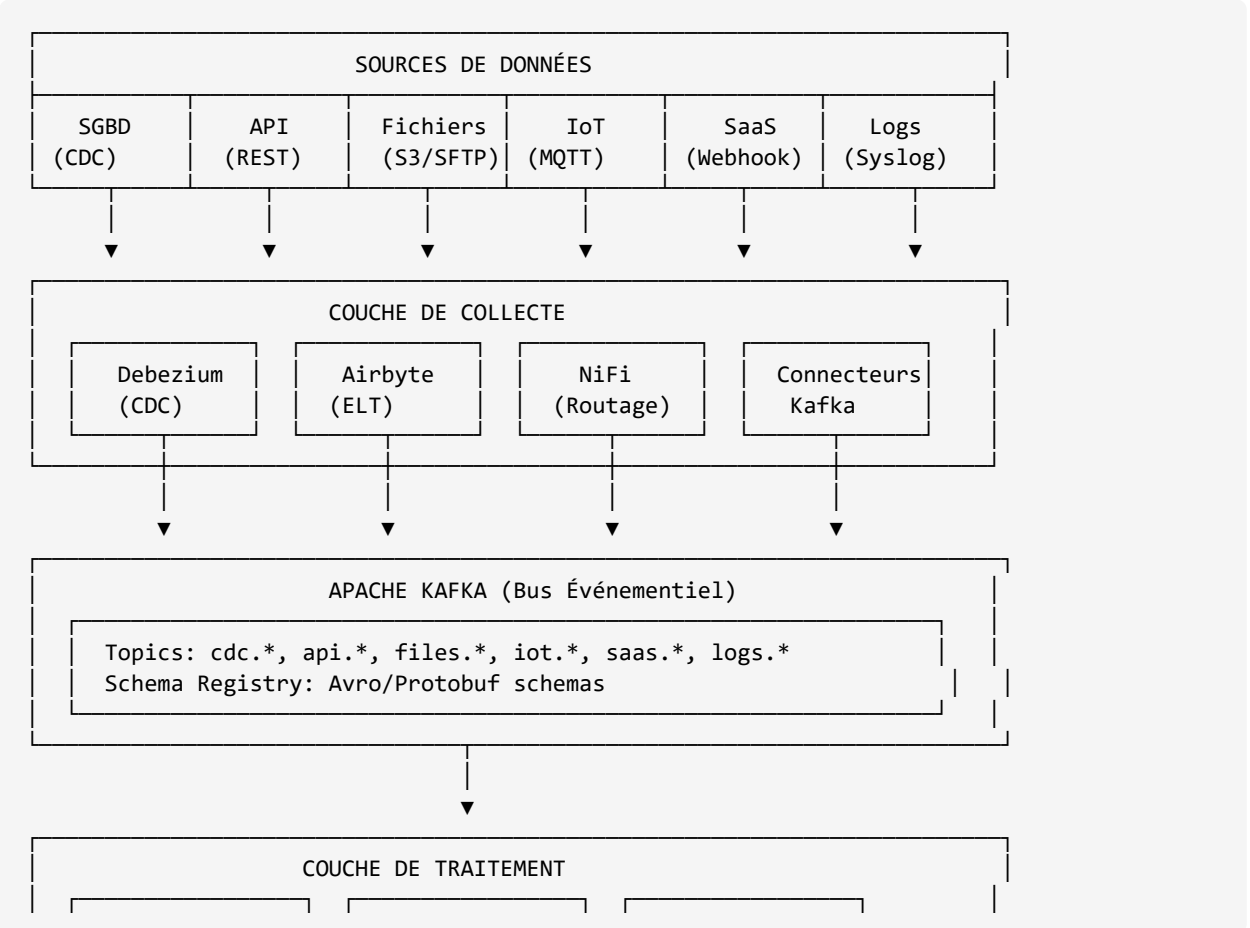
Apache Iceberg, combiné à Kafka et un moteur de traitement adéquat (Flink, Spark Structured Streaming), permet d'atteindre la sémantique exactly-once grâce à :

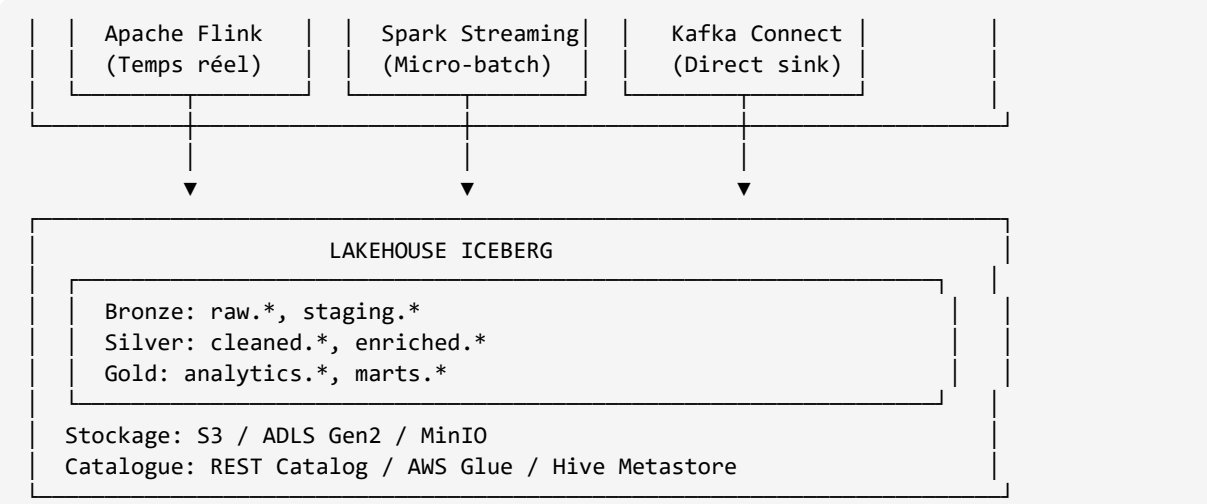
- La gestion transactionnelle des commits Iceberg
- Les offsets Kafka comme marqueurs de progression
- Le checkpointing des moteurs de traitement

Architecture de Référence

Vue d'Ensemble

Une architecture d'ingestion complète pour un Lakehouse Iceberg comprend plusieurs composantes orchestrées pour garantir fiabilité, performance et observabilité.





Composantes Clés

Couche de Collecte : Responsable de l'extraction des données depuis les systèmes sources. Cette couche doit gérer la diversité des protocoles (JDBC, REST, SFTP, MQTT), la transformation initiale des formats et la gestion des erreurs de connexion.

Bus Événementiel (Kafka) : Point central de découplage entre les producteurs et les consommateurs. Kafka assure la persistance temporaire, le replay en cas d'erreur et la distribution vers de multiples consommateurs. Le Schema Registry garantit la compatibilité des schémas.

Couche de Traitement : Transforme et charge les données dans Iceberg. Le choix du moteur dépend des exigences de latence : Flink pour le temps réel, Spark Structured Streaming pour le micro-batch, Kafka Connect pour les chargements directs simples.

Lakehouse Iceberg : Destination finale structurée en couches médaillon (Bronze/Silver/Gold) pour organiser les données selon leur niveau de raffinement.

Dimensionnement Initial

Le dimensionnement de l'architecture d'ingestion dépend de plusieurs facteurs interdépendants :

Facteur	Impact	Recommandation initiale
Volume quotidien	Capacité Kafka, stockage	3× le volume quotidien en rétention Kafka
Pic de débit	Partitions Kafka, workers	Prévoir 2× le débit moyen
Nombre de sources	Connecteurs, parallélisme	1 topic par source majeure
Latence cible	Choix du moteur	Flink < 1s, Spark 1-15min
Fenêtre de replay	Rétention Kafka	7 jours minimum

Ingestion Batch avec Apache Spark

Configuration Spark-Iceberg

Apache Spark demeure le moteur de référence pour l'ingestion batch dans Iceberg. Sa maturité, son écosystème riche et son intégration native avec Iceberg en font le choix par défaut pour les chargements volumineux.

Configuration de base :

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("IngestionBatchIceberg") \
    .config("spark.sql.extensions",
"org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions") \
    .config("spark.sql.catalog.lakehouse", "org.apache.iceberg.spark.SparkCatalog") \
    .config("spark.sql.catalog.lakehouse.type", "rest") \
    .config("spark.sql.catalog.lakehouse.uri", "http://iceberg-rest-catalog:8181") \
    .config("spark.sql.catalog.lakehouse.warehouse", "s3://entreprise-lakehouse/
warehouse") \
    .config("spark.sql.catalog.lakehouse.io-impl", "org.apache.iceberg.aws.s3.S3FileIO") \
    .config("spark.hadoop.fs.s3a.endpoint", "s3.ca-central-1.amazonaws.com") \
    .config("spark.hadoop.fs.s3a.aws.credentials.provider",
            "com.amazonaws.auth.DefaultAWSCredentialsProviderChain") \
    .getOrCreate()
```

Paramètres de performance critiques :

```
# Optimisations pour l'écriture batch
spark.conf.set("spark.sql.shuffle.partitions", "200")
spark.conf.set("spark.sql.adaptive.enabled", "true")
spark.conf.set("spark.sql.adaptive.coalescePartitions.enabled", "true")

# Configuration Iceberg spécifique
spark.conf.set("spark.sql.catalog.lakehouse.write.target-file-size-bytes", "536870912")  #
512 Mo
spark.conf.set("spark.sql.catalog.lakehouse.write.distribution-mode", "hash")
```

Patterns d'Écriture Batch

Append simple : Le pattern le plus performant pour les données nouvelles sans mise à jour.

```
# Lecture depuis la source
df_nouvelles_transactions = spark.read \
    .format("jdbc") \
    .option("url", "jdbc:postgresql://source-db:5432/production") \
    .option("dbtable", "(SELECT * FROM transactions WHERE date_creation >= '2024-01-15')
AS t") \
    .option("user", "reader") \
    .option("password", "${DB_PASSWORD}") \
    .option("fetchsize", "10000") \
    .option("numPartitions", "16") \
    .option("partitionColumn", "id") \
```

```
    .option("lowerBound", "1") \
    .option("upperBound", "10000000") \
    .load()

# Écriture en mode append
df_nouvelles_transactions.writeTo("lakehouse.bronze.transactions") \
    .option("write.distribution-mode", "hash") \
    .option("write.target-file-size-bytes", "536870912") \
    .append()
```

Chargement initial complet : Pour les migrations ou les premières charges.

```
# Création de la table avec partitionnement
spark.sql("""
    CREATE TABLE IF NOT EXISTS lakehouse.bronze.clients (
        client_id BIGINT,
        nom STRING,
        courriel STRING,
        date_inscription DATE,
        province STRING,
        segment STRING
    )
    USING iceberg
    PARTITIONED BY (province, months(date_inscription))
    TBLPROPERTIES (
        'write.target-file-size-bytes' = '536870912',
        'write.distribution-mode' = 'hash',
        'format-version' = '2'
    )
""")

# Chargement avec repartitionnement optimal
df_clients = spark.read.format("jdbc") \
    .option("url", "jdbc:oracle:thin:@legacy-crm:1521/PROD") \
    .option("dbtable", "CLIENTS") \
    .load()

df_clients \
    .repartition(100, "province") \
    .sortWithinPartitions("date_inscription") \
    .writeTo("lakehouse.bronze.clients") \
    .createOrReplace()
```

Optimisation des Extractions JDBC

L'extraction depuis les bases de données relationnelles constitue souvent le goulot d'étranglement de l'ingestion batch. Plusieurs stratégies permettent d'optimiser ce processus.

Parallélisation de l'extraction :

```
# Extraction parallèle avec partitionnement numérique
df = spark.read.format("jdbc") \
    .option("url", jdbc_url) \
    .option("dbtable", "commandes") \
    .option("numPartitions", 32) \
```

```
    .option("partitionColumn", "commande_id") \
    .option("lowerBound", 1) \
    .option("upperBound", max_id) \
    .option("fetchsize", 10000) \
    .load()

# Alternative: partitionnement par prédicat pour colonnes non numériques
predicates = [
    "province = 'QC'",
    "province = 'ON'",
    "province = 'BC'",
    "province = 'AB'",
    "province NOT IN ('QC', 'ON', 'BC', 'AB')"
]

df = spark.read.jdbc(
    url=jdbc_url,
    table="commandes",
    predicates=predicates,
    properties={"fetchsize": "10000"}
)
```

Extraction incrémentale avec watermark :

```
from datetime import datetime, timedelta

# Récupération du dernier watermark
dernier_watermark = spark.sql("""
    SELECT MAX(date_modification) as watermark
    FROM lakehouse.bronze.commandes
""").collect()[0]["watermark"]

# Extraction des nouvelles données uniquement
df_incremental = spark.read.format("jdbc") \
    .option("url", jdbc_url) \
    .option("dbtable", f"""
        (SELECT * FROM commandes
         WHERE date_modification > '{dernier_watermark}'
         AND date_modification <= '{datetime.now()}') AS incr
    """) \
    .load()

# Écriture avec mise à jour du watermark
df_incremental.writeTo("lakehouse.bronze.commandes").append()
```

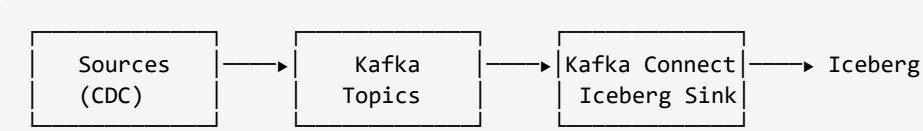
> **Performance**
>
> L'extraction JDBC parallèle peut améliorer les temps de chargement de 5× à 20× selon la configuration. Cependant, attention à ne pas surcharger le système source : limitez le nombre de connexions parallèles et planifiez les extractions durant les périodes de faible activité.

Ingestion Streaming avec Apache Kafka

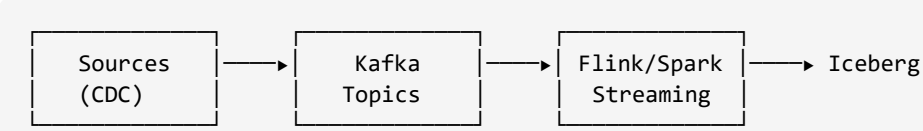
Architecture Kafka-Iceberg

L'intégration Kafka-Iceberg forme le cœur du Streaming Lakehouse. Deux approches principales permettent de connecter ces technologies :

Approche 1 : Kafka Connect avec Iceberg Sink



Approche 2 : Moteur de Traitement Streaming



Le choix entre ces approches dépend de la complexité des transformations requises. Kafka Connect convient aux chargements directs avec transformations minimales, tandis qu'un moteur de traitement s'impose pour les agrégations, jointures ou logiques métier complexes.

Configuration Kafka Connect Iceberg Sink

Le connecteur Iceberg Sink pour Kafka Connect, développé par Tabular (maintenant partie de Databricks), offre une solution clé en main pour l'ingestion streaming.

```
{
  "name": "iceberg-sink-transactions",
  "config": {
    "connector.class": "io.tabular.iceberg.connect.IcebergSinkConnector",
    "tasks.max": "4",
    "topics": "cdc.production.transactions",

    "iceberg.tables": "lakehouse.bronze.transactions",
    "iceberg.catalog.type": "rest",
    "iceberg.catalog.uri": "http://iceberg-rest-catalog:8181",
    "iceberg.catalog.warehouse": "s3://entreprise-lakehouse/warehouse",
    "iceberg.catalog.s3.endpoint": "s3.ca-central-1.amazonaws.com",

    "iceberg.table.auto-create": "true",
    "iceberg.table.evolve-schema-enabled": "true",

    "iceberg.control.commit.interval-ms": "60000",
    "iceberg.control.commit.timeout-ms": "300000",

    "key.converter": "io.confluent.connect.avro.AvroConverter",
    "key.converter.schema.registry.url": "http://schema-registry:8081",
    "value.converter": "io.confluent.connect.avro.AvroConverter",
    "value.converter.schema.registry.url": "http://schema-registry:8081",
```

```
    "transforms": "unwrap",
    "transforms.unwrap.type": "io.debezium.transforms.ExtractNewRecordState",
    "transforms.unwrap.drop.tombstones": "false",
    "transforms.unwrap.delete.handling.mode": "rewrite"
  }
}
```

Paramètres critiques :

Paramètre	Description	Recommandation
`tasks.max`	Parallélisme du connecteur	1 par partition Kafka (max)
`commit.interval-ms`	Fréquence des commits Iceberg	60-300s selon latence cible
`commit.timeout-ms`	Timeout pour commit	3-5× l'intervalle
`evolve-schema-enabled`	Évolution automatique	`true` pour CDC

Apache Flink pour le Streaming Temps Réel

Apache Flink excelle pour les scénarios nécessitant une latence sub-seconde ou des transformations complexes en streaming. Son intégration native avec Iceberg via le connecteur Flink-Iceberg offre des garanties exactly-once robustes.

Configuration Flink-Iceberg :

```
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.enableCheckpointing(60000); // Checkpoint toutes les 60 secondes

StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

// Configuration du catalogue Iceberg
tableEnv.executeSql("""
    CREATE CATALOG lakehouse WITH (
        'type' = 'iceberg',
        'catalog-type' = 'rest',
        'uri' = 'http://iceberg-rest-catalog:8181',
        'warehouse' = 's3://entreprise-lakehouse/warehouse',
        'io-impl' = 'org.apache.iceberg.aws.s3.S3FileIO'
    )
""");

// Définition de la source Kafka
tableEnv.executeSql("""
    CREATE TABLE kafka_transactions (
        transaction_id BIGINT,
        client_id BIGINT,
        montant DECIMAL(10,2),
        devise STRING,
        timestamp_evenement TIMESTAMP(3),
        WATERMARK FOR timestamp_evenement AS timestamp_evenement - INTERVAL '5' SECOND
    ) WITH (
```

```
        'connector' = 'kafka',
        'topic' = 'cdc.production.transactions',
        'properties.bootstrap.servers' = 'kafka:9092',
        'properties.group.id' = 'flink-iceberg-ingestion',
        'scan.startup.mode' = 'earliest-offset',
        'format' = 'avro-confluent',
        'avro-confluent.url' = 'http://schema-registry:8081'
    )
""");

// Ingestion continue vers Iceberg
tableEnv.executeSql("""
    INSERT INTO lakehouse.bronze.transactions
    SELECT
        transaction_id,
        client_id,
        montant,
        devise,
        timestamp_evenement,
        CURRENT_TIMESTAMP as timestamp_ingestion
    FROM kafka_transactions
""");
```

Agrégations en temps réel :

```
-- Table d'agrégation par fenêtre temporelle
CREATE TABLE lakehouse.silver.metriques_temps_reel (
    fenetre_debut TIMESTAMP,
    fenetre_fin TIMESTAMP,
    total_transactions BIGINT,
    montant_total DECIMAL(15,2),
    montant_moyen DECIMAL(10,2)
) WITH (
    'format-version' = '2',
    'write.upsert.enabled' = 'true'
);

-- Pipeline d'agrégation
INSERT INTO lakehouse.silver.metriques_temps_reel
SELECT
    window_start as fenetre_debut,
    window_end as fenetre_fin,
    COUNT(*) as total_transactions,
    SUM(montant) as montant_total,
    AVG(montant) as montant_moyen
FROM TABLE(
    TUMBLE(TABLE kafka_transactions, DESCRIPTOR(timestamp_evenement), INTERVAL '1' MINUTE)
)
GROUP BY window_start, window_end;
```

Spark Structured Streaming

Pour les organisations déjà investies dans l'écosystème Spark, Structured Streaming offre une alternative mature à Flink avec une courbe d'apprentissage réduite.

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import from_json, col, current_timestamp
from pyspark.sql.types import StructType, StructField, LongType, DecimalType, StringType,
TimestampType

# Schéma des événements
schema_transaction = StructType([
    StructField("transaction_id", LongType()),
    StructField("client_id", LongType()),
    StructField("montant", DecimalType(10, 2)),
    StructField("devise", StringType()),
    StructField("timestamp_evenement", TimestampType())
])

# Lecture streaming depuis Kafka
df_stream = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "kafka:9092") \
    .option("subscribe", "cdc.production.transactions") \
    .option("startingOffsets", "earliest") \
    .option("maxOffsetsPerTrigger", 100000) \
    .load()

# Transformation
df_transactions = df_stream \
    .select(from_json(col("value").cast("string"), schema_transaction).alias("data")) \
    .select("data.*") \
    .withColumn("timestamp_ingestion", current_timestamp())

# Écriture vers Iceberg
query = df_transactions.writeStream \
    .format("iceberg") \
    .outputMode("append") \
    .option("path", "lakehouse.bronze.transactions") \
    .option("checkpointLocation", "s3://entreprise-lakehouse/checkpoints/transactions") \
    .trigger(processingTime="1 minute") \
    .start()
```

Performance

Spark Structured Streaming avec Iceberg atteint typiquement des latences de 1 à 5 minutes selon la configuration du trigger. Pour des latences sub-minute, privilégiez Apache Flink ou Kafka Connect avec des intervalles de commit courts.

Change Data Capture (CDC)

Fondamentaux du CDC

Le Change Data Capture représente la technique privilégiée pour synchroniser les données des systèmes transactionnels vers le Lakehouse. Plutôt que d'extraire périodiquement l'intégralité des tables, le CDC

capture uniquement les modifications — insertions, mises à jour, suppressions — réduisant drastiquement le volume de données transférées et la charge sur les systèmes sources.

Types de CDC :

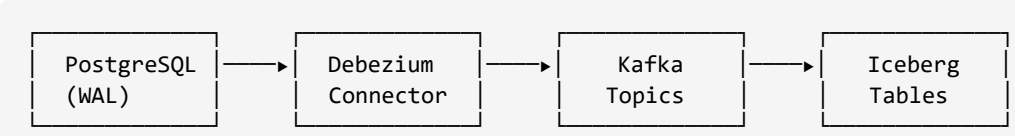
Type	Mécanisme	Latence	Impact source
Log-based	Lecture des journaux de transactions	Secondes	Minimal
Trigger-based	Déclencheurs base de données	Secondes	Modéré
Query-based	Requêtes périodiques sur timestamp	Minutes	Variable
Timestamp-based	Colonne de modification	Minutes	Minimal

Le CDC basé sur les journaux (log-based) offre le meilleur compromis latence/impact et constitue la recommandation par défaut pour les bases de données le supportant.

Debezium : La Référence Open Source

Debezium s'est imposé comme la solution CDC de référence dans l'écosystème open source. Fonctionnant comme connecteur Kafka Connect, il capture les modifications depuis les journaux de transaction des principales bases de données.

Architecture Debezium-Kafka-Iceberg :



Configuration Debezium pour PostgreSQL :

```
{
  "name": "postgres-cdc-source",
  "config": {
    "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
    "tasks.max": "1",

    "database.hostname": "postgres-production.internal",
    "database.port": "5432",
    "database.user": "debezium",
    "database.password": "${POSTGRES_PASSWORD}",
    "database.dbname": "production",
    "database.server.name": "prod",

    "plugin.name": "pgoutput",
    "publication.name": "debezium_publication",
    "slot.name": "debezium_slot",

    "table.include.list": "public.clients,public.commandes,public.produits",

    "key.converter": "io.confluent.connect.avro.AvroConverter",
    "key.converter.schema.registry.url": "http://schema-registry:8081",
    "value.converter": "io.confluent.connect.avro.AvroConverter",
    "value.converter.schema.registry.url": "http://schema-registry:8081",
```

```
    "transforms": "route",
    "transforms.route.type": "org.apache.kafka.connect.transforms.RegexRouter",
    "transforms.route.regex": "prod\\.public\\.(.*)",
    "transforms.route.replacement": "cdc.production.$1",

    "heartbeat.interval.ms": "10000",
    "snapshot.mode": "initial"
  }
}
```

Traitement des Événements CDC dans Iceberg

Les événements CDC contiennent des métadonnées indiquant le type d'opération (c=create, u=update, d=delete) et l'état avant/après de l'enregistrement. Plusieurs stratégies permettent de matérialiser ces événements dans Iceberg.

Stratégie 1 : Append-only avec historisation

Conserve tous les événements bruts, permettant une reconstruction complète de l'historique.

```
# Lecture des événements CDC
df_cdc = spark.readStream \
    .format("kafka") \
    .option("subscribe", "cdc.production.clients") \
    .load()

# Extraction des champs Debezium
df_parsed = df_cdc.select(
    from_json(col("value").cast("string"), schema_debezium).alias("data")
).select(
    col("data.after.*"),
    col("data.op").alias("operation"),
    col("data.ts_ms").alias("timestamp_cdc"),
    current_timestamp().alias("timestamp_ingestion")
)

# Écriture append-only
df_parsed.writeStream \
    .format("iceberg") \
    .outputMode("append") \
    .option("path", "lakehouse.bronze.clients_cdc") \
    .start()
```

Stratégie 2 : Upsert avec état courant

Maintient une vue à jour de l'état courant via les opérations MERGE d'Iceberg.

```
from pyspark.sql.functions import window, max as spark_max

# Micro-batch avec déduplication
def process_batch(batch_df, batch_id):
    # Garder uniquement le dernier état par clé
    latest = batch_df \
        .groupBy("client_id") \
        .agg(spark_max("timestamp_cdc").alias("max_ts")) \
```

```
        .join(batch_df, ["client_id"]) \
        .where(col("timestamp_cdc") == col("max_ts"))

    # MERGE INTO pour upsert
    latest.createOrReplaceTempView("updates")

    spark.sql("""
        MERGE INTO lakehouse.silver.clients t
        USING updates s
        ON t.client_id = s.client_id
        WHEN MATCHED AND s.operation = 'd' THEN DELETE
        WHEN MATCHED THEN UPDATE SET *
        WHEN NOT MATCHED AND s.operation != 'd' THEN INSERT *
    """)

df_cdc.writeStream \
    .foreachBatch(process_batch) \
    .option("checkpointLocation", checkpoint_path) \
    .trigger(processingTime="5 minutes") \
    .start()
```

Migration

De : Réplication batch quotidienne avec comparaison de tables

Vers : CDC temps réel avec Debezium-Kafka-Iceberg

Stratégie : Déploiement progressif table par table, en commençant par les tables les moins critiques. Validation par comparaison batch/CDC pendant 2 semaines avant basculement complet.

Gestion des Suppressions

Les suppressions méritent une attention particulière dans les architectures Lakehouse. Trois approches coexistent :

Suppression physique (hard delete) : L'enregistrement est réellement supprimé de la table Iceberg. Simple mais irréversible et potentiellement problématique pour l'audit.

```
MERGE INTO lakehouse.silver.clients t
USING deletes s
ON t.client_id = s.client_id
WHEN MATCHED THEN DELETE
```

Suppression logique (soft delete) : Un indicateur marque l'enregistrement comme supprimé sans le retirer physiquement.

```
MERGE INTO lakehouse.silver.clients t
USING deletes s
ON t.client_id = s.client_id
WHEN MATCHED THEN UPDATE SET
    est_supprime = true,
    date_suppression = current_timestamp()
```

Historisation complète : Conserve toutes les versions avec dates d'effet (SCD Type 2).

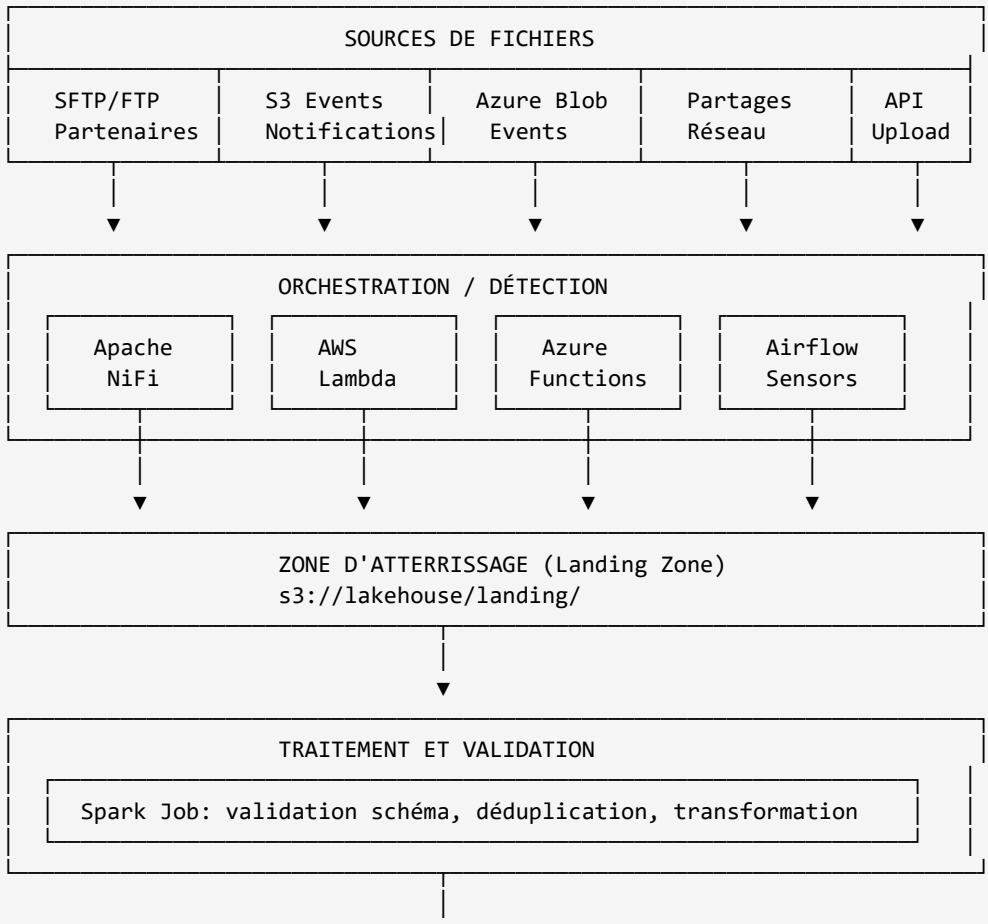
```sql
-- Fermeture de la version précédente
UPDATE lakehouse.silver.clients_historique
SET date_fin_validite = current_timestamp(), est_courant = false
WHERE client_id IN (SELECT client_id FROM deletes)
  AND est_courant = true;

-- Insertion de la version supprimée
INSERT INTO lakehouse.silver.clients_historique
SELECT *, current_timestamp() as date_debut_validite,
       null as date_fin_validite, true as est_courant, 'SUPPRIME' as statut
FROM deletes;
```

Ingestion de Fichiers

Patterns d'Ingestion de Fichiers

L'ingestion de fichiers demeure un pattern incontournable pour les échanges avec partenaires externes, les exports de systèmes legacy et les données semi-structurées. Plusieurs mécanismes permettent d'automatiser ce processus.

Architecture d'ingestion de fichiers :



```
                                ▼
┌──────────────────────────────────────────────────────────────────────────┐
│                          LAKEHOUSE ICEBERG                               │
│                      lakehouse.bronze.fichiers_*                         │
└──────────────────────────────────────────────────────────────────────────┘
```

Détection et Déclenchement

S3 Event Notifications avec Lambda :

```python
# Lambda function déclenchée par S3 Event
import boto3
import json

def lambda_handler(event, context):
    # Extraction des informations du fichier
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = event['Records'][0]['s3']['object']['key']

    # Publication vers Kafka pour traitement
    kafka_client = boto3.client('kafka')

    message = {
        'source': 's3',
        'bucket': bucket,
        'key': key,
        'timestamp': event['Records'][0]['eventTime'],
        'size': event['Records'][0]['s3']['object']['size']
    }

    # Déclenchement du pipeline d'ingestion
    # Via Step Functions, Airflow, ou publication Kafka
    trigger_ingestion_pipeline(message)

    return {'statusCode': 200}
```

Apache Airflow avec File Sensor :

```python
from airflow import DAG
from airflow.providers.amazon.aws.sensors.s3 import S3KeySensor
from airflow.providers.apache.spark.operators.spark_submit import SparkSubmitOperator
from datetime import datetime, timedelta

default_args = {
    'owner': 'data-platform',
    'retries': 3,
    'retry_delay': timedelta(minutes=5)
}

with DAG(
    'ingestion_fichiers_partenaires',
    default_args=default_args,
    schedule_interval='*/15 * * * *',  # Toutes les 15 minutes
    catchup=False
) as dag:
```

```
    # Détection de nouveaux fichiers
    detect_fichiers = S3KeySensor(
        task_id='detect_nouveaux_fichiers',
        bucket_name='entreprise-landing',
        bucket_key='partenaires/{{ ds }}/*.csv',
        wildcard_match=True,
        timeout=60 * 10,
        poke_interval=60
    )

    # Traitement Spark
    traiter_fichiers = SparkSubmitOperator(
        task_id='traiter_fichiers_csv',
        application='s3://entreprise-lakehouse/jobs/ingestion_csv.py',
        conf={
            'spark.sql.catalog.lakehouse': 'org.apache.iceberg.spark.SparkCatalog',
            'spark.sql.catalog.lakehouse.type': 'rest'
        },
        application_args=['--date', '{{ ds }}', '--source', 'partenaires']
    )

    detect_fichiers >> traiter_fichiers
```

Traitement de Formats Variés

CSV avec inférence et validation de schéma :

```
from pyspark.sql.functions import input_file_name, current_timestamp
from pyspark.sql.types import StructType, StructField, StringType, DecimalType, DateType

# Schéma attendu (contrat avec le partenaire)
schema_attendu = StructType([
    StructField("code_produit", StringType(), False),
    StructField("description", StringType(), True),
    StructField("prix_unitaire", DecimalType(10, 2), False),
    StructField("date_effet", DateType(), False)
])

# Lecture avec validation
df_fichiers = spark.read \
    .option("header", "true") \
    .option("delimiter", ";") \
    .option("encoding", "UTF-8") \
    .option("mode", "PERMISSIVE") \
    .option("columnNameOfCorruptRecord", "_corrupt_record") \
    .schema(schema_attendu) \
    .csv("s3://entreprise-landing/partenaires/2024-01-15/*.csv")

# Ajout de métadonnées de traçabilité
df_enrichi = df_fichiers \
    .withColumn("fichier_source", input_file_name()) \
    .withColumn("timestamp_ingestion", current_timestamp())

# Séparation données valides / invalides
df_valides = df_enrichi.filter(col("_corrupt_record").isNull())
df_erreurs = df_enrichi.filter(col("_corrupt_record").isNotNull())
```

```
# Écriture des données valides
df_valides.drop("_corrupt_record") \
    .writeTo("lakehouse.bronze.produits_partenaire") \
    .append()

# Archivage des erreurs pour analyse
df_erreurs.writeTo("lakehouse.quarantine.produits_erreurs").append()
```

JSON semi-structuré avec schéma évolutif :

```
from pyspark.sql.functions import explode, get_json_object

# Lecture JSON avec inférence de schéma
df_json = spark.read \
    .option("multiLine", "true") \
    .option("mode", "PERMISSIVE") \
    .json("s3://entreprise-landing/api-exports/2024-01-15/")

# Aplatissement de structures imbriquées
df_aplati = df_json \
    .select(
        col("transaction_id"),
        col("client.id").alias("client_id"),
        col("client.nom").alias("client_nom"),
        explode("lignes_commande").alias("ligne")
    ) \
    .select(
        col("transaction_id"),
        col("client_id"),
        col("client_nom"),
        col("ligne.produit_id"),
        col("ligne.quantite"),
        col("ligne.prix_unitaire")
    )

# Écriture avec évolution de schéma activée
df_aplati.writeTo("lakehouse.bronze.transactions_api") \
    .option("merge-schema", "true") \
    .append()
```

Qualité des Données à l'Ingestion

Validation en Ligne

L'intégration de contrôles de qualité directement dans les pipelines d'ingestion permet de détecter les anomalies au plus tôt, réduisant les coûts de correction et protégeant l'intégrité du Lakehouse.

Framework de validation avec Great Expectations :

```
import great_expectations as gx
from great_expectations.core.batch import RuntimeBatchRequest

# Configuration du contexte
context = gx.get_context()

# Définition des attentes pour les transactions
expectation_suite = context.create_expectation_suite(
    expectation_suite_name="transactions_ingestion",
    overwrite_existing=True
)

# Attentes de base
expectation_suite.add_expectation(
    gx.core.ExpectationConfiguration(
        expectation_type="expect_column_values_to_not_be_null",
        kwargs={"column": "transaction_id"}
    )
)

expectation_suite.add_expectation(
    gx.core.ExpectationConfiguration(
        expectation_type="expect_column_values_to_be_between",
        kwargs={"column": "montant", "min_value": 0, "max_value": 1000000}
    )
)

expectation_suite.add_expectation(
    gx.core.ExpectationConfiguration(
        expectation_type="expect_column_values_to_match_regex",
        kwargs={"column": "courriel", "regex": r"^[\w\.-]+@[\w\.-]+\.\w+$"}
    )
)

# Validation dans le pipeline Spark
def validate_batch(batch_df, batch_id):
    # Conversion en DataFrame Pandas pour GX
    pandas_df = batch_df.toPandas()

    batch_request = RuntimeBatchRequest(
        datasource_name="spark_datasource",
        data_connector_name="runtime_data_connector",
        data_asset_name="transactions_batch",
        runtime_parameters={"batch_data": pandas_df},
        batch_identifiers={"batch_id": str(batch_id)}
    )

    validator = context.get_validator(
        batch_request=batch_request,
        expectation_suite_name="transactions_ingestion"
    )

    results = validator.validate()

    if results.success:
        # Écriture dans la table principale
        batch_df.writeTo("lakehouse.bronze.transactions").append()
    else:
        # Quarantaine des données invalides
```

```
        batch_df.writeTo("lakehouse.quarantine.transactions").append()
        # Alerte
        send_quality_alert(results, batch_id)
```

Déduplication

La déduplication à l'ingestion prévient l'accumulation de doublons qui dégradent les analyses et augmentent les coûts de stockage.

Déduplication en streaming :

```
from pyspark.sql.functions import window, row_number
from pyspark.sql.window import Window

def deduplicate_stream(df):
    # Fenêtre de déduplication: garde le premier enregistrement par clé
    window_spec = Window \
        .partitionBy("transaction_id") \
        .orderBy(col("timestamp_evenement").asc())

    df_dedup = df \
        .withColumn("row_num", row_number().over(window_spec)) \
        .filter(col("row_num") == 1) \
        .drop("row_num")

    return df_dedup

# Application dans le pipeline streaming
df_stream \
    .transform(deduplicate_stream) \
    .writeStream \
    .format("iceberg") \
    .option("path", "lakehouse.bronze.transactions") \
    .start()
```

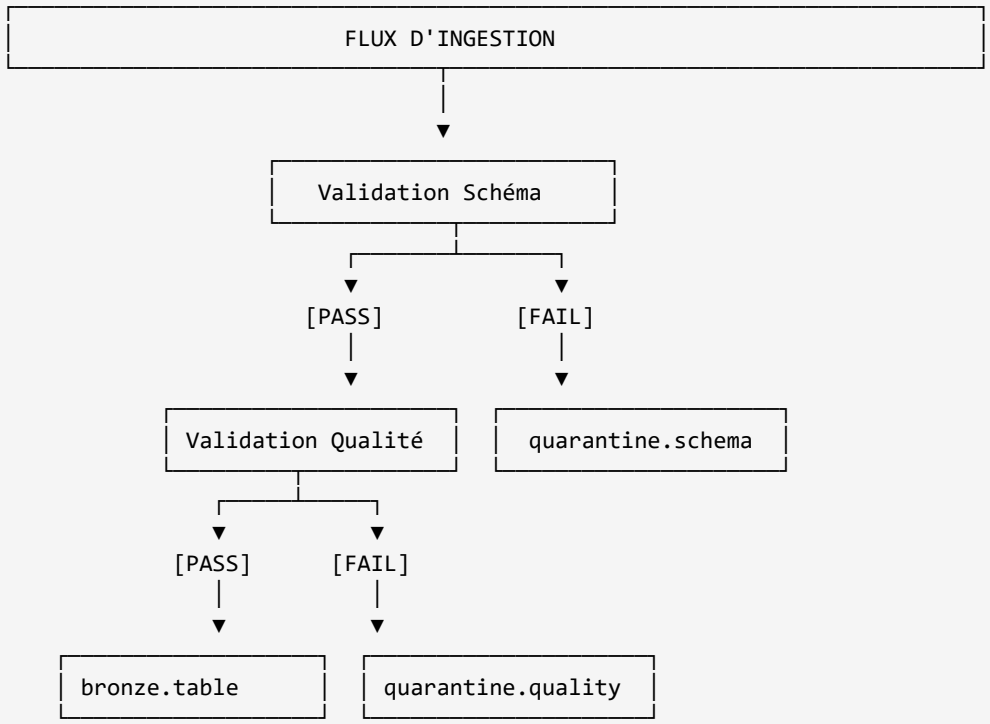
Déduplication batch avec Iceberg MERGE :

```
-- Déduplication lors de l'insertion
MERGE INTO lakehouse.bronze.transactions t
USING (
    SELECT * FROM (
        SELECT *, ROW_NUMBER() OVER (
            PARTITION BY transaction_id
            ORDER BY timestamp_evenement DESC
        ) as rn
        FROM staging.nouvelles_transactions
    ) WHERE rn = 1
) s
ON t.transaction_id = s.transaction_id
WHEN NOT MATCHED THEN INSERT *;
```

Gestion des Erreurs et Quarantaine

Une stratégie robuste de gestion des erreurs distingue les pipelines de production fiables des prototypes fragiles.

Architecture de quarantaine :



Implémentation de la quarantaine :

```python
from dataclasses import dataclass
from enum import Enum
from datetime import datetime

class ErreurType(Enum):
    SCHEMA = "schema_mismatch"
    QUALITE = "quality_failure"
    TRANSFORMATION = "transformation_error"
    TIMEOUT = "processing_timeout"

@dataclass
class EnregistrementQuarantaine:
    donnees_originales: str
    type_erreur: ErreurType
    message_erreur: str
    source: str
    timestamp_erreur: datetime
    tentatives: int

def process_with_quarantine(batch_df, batch_id):
    try:
        # Validation schéma
        df_valide_schema = validate_schema(batch_df)
        df_erreurs_schema = get_schema_errors(batch_df)

        # Quarantaine des erreurs de schéma
        if df_erreurs_schema.count() > 0:
            quarantine_records(df_erreurs_schema, ErreurType.SCHEMA)
```

```
        # Validation qualité
        df_valide_qualite = validate_quality(df_valide_schema)
        df_erreurs_qualite = get_quality_errors(df_valide_schema)

        # Quarantaine des erreurs de qualité
        if df_erreurs_qualite.count() > 0:
            quarantine_records(df_erreurs_qualite, ErreurType.QUALITE)

        # Écriture des données valides
        df_valide_qualite.writeTo("lakehouse.bronze.transactions").append()

        # Métriques
        log_ingestion_metrics(batch_id,
                              total=batch_df.count(),
                              valides=df_valide_qualite.count(),
                              quarantaine=df_erreurs_schema.count() +
df_erreurs_qualite.count())

    except Exception as e:
        # Quarantaine complète du batch en cas d'erreur inattendue
        quarantine_entire_batch(batch_df, ErreurType.TRANSFORMATION, str(e))
        raise

def quarantine_records(df_erreurs, type_erreur):
    df_quarantaine = df_erreurs \
        .withColumn("type_erreur", lit(type_erreur.value)) \
        .withColumn("timestamp_quarantaine", current_timestamp()) \
        .withColumn("batch_id", lit(batch_id))

    df_quarantaine.writeTo("lakehouse.quarantine.transactions").append()
```

Orchestration des Pipelines

Apache Airflow pour l'Orchestration

Apache Airflow demeure le standard de facto pour l'orchestration des pipelines de données complexes. Son modèle de DAG (Directed Acyclic Graph) permet d'exprimer les dépendances entre tâches et de gérer les reprises sur erreur.

DAG d'ingestion complète :

```
from airflow import DAG
from airflow.operators.python import PythonOperator, BranchPythonOperator
from airflow.providers.apache.spark.operators.spark_submit import SparkSubmitOperator
from airflow.providers.slack.operators.slack_webhook import SlackWebhookOperator
from airflow.utils.task_group import TaskGroup
from datetime import datetime, timedelta

default_args = {
    'owner': 'data-engineering',
    'depends_on_past': False,
    'email_on_failure': True,
```

```
    'email': ['data-alerts@entreprise.ca'],
    'retries': 3,
    'retry_delay': timedelta(minutes=5),
    'retry_exponential_backoff': True
}

with DAG(
    'ingestion_quotidienne_ventes',
    default_args=default_args,
    description='Pipeline d\'ingestion des données de ventes',
    schedule_interval='0 6 * * *',  # 6h00 quotidien
    start_date=datetime(2024, 1, 1),
    catchup=False,
    tags=['ingestion', 'ventes', 'production']
) as dag:

    # Vérification de la disponibilité des sources
    with TaskGroup('verification_sources') as verification:
        verif_postgres = PythonOperator(
            task_id='verif_postgres',
            python_callable=check_postgres_availability
        )

        verif_api = PythonOperator(
            task_id='verif_api',
            python_callable=check_api_availability
        )

    # Extraction parallèle
    with TaskGroup('extraction') as extraction:
        extract_transactions = SparkSubmitOperator(
            task_id='extract_transactions',
            application='s3://lakehouse/jobs/extract_transactions.py',
            conf={'spark.executor.instances': '10'},
            application_args=['--date', '{{ ds }}']
        )

        extract_clients = SparkSubmitOperator(
            task_id='extract_clients',
            application='s3://lakehouse/jobs/extract_clients.py',
            application_args=['--date', '{{ ds }}']
        )

        extract_produits = SparkSubmitOperator(
            task_id='extract_produits',
            application='s3://lakehouse/jobs/extract_produits.py',
            application_args=['--date', '{{ ds }}']
        )

    # Validation qualité
    validation = SparkSubmitOperator(
        task_id='validation_qualite',
        application='s3://lakehouse/jobs/validate_ingestion.py',
        application_args=['--date', '{{ ds }}']
    )

    # Branchement selon résultats validation
    def decide_next_step(**context):
        validation_result = context['ti'].xcom_pull(task_ids='validation_qualite')
```

```
        if validation_result['success_rate'] > 0.99:
            return 'transformation'
        else:
            return 'alerte_qualite'

    branch = BranchPythonOperator(
        task_id='branch_validation',
        python_callable=decide_next_step
    )

    # Transformation Silver
    transformation = SparkSubmitOperator(
        task_id='transformation',
        application='s3://lakehouse/jobs/transform_silver.py',
        application_args=['--date', '{{ ds }}']
    )

    # Alerte en cas de problème
    alerte = SlackWebhookOperator(
        task_id='alerte_qualite',
        slack_webhook_conn_id='slack_data_alerts',
        message='⚠️ Qualité des données insuffisante pour {{ ds }}'
    )

    # Dépendances
    verification >> extraction >> validation >> branch
    branch >> [transformation, alerte]
```

Patterns d'Orchestration Avancés

Ingestion incrémentale avec gestion d'état :

```
from airflow.models import Variable
from airflow.operators.python import PythonOperator

def get_watermark(**context):
    """Récupère le dernier watermark traité"""
    watermark = Variable.get(
        'watermark_transactions',
        default_var='1970-01-01T00:00:00Z'
    )
    context['ti'].xcom_push(key='watermark', value=watermark)
    return watermark

def update_watermark(**context):
    """Met à jour le watermark après traitement réussi"""
    new_watermark = context['ti'].xcom_pull(
        task_ids='extract_incremental',
        key='max_timestamp'
    )
    Variable.set('watermark_transactions', new_watermark)

with DAG('ingestion_incrementale') as dag:

    get_wm = PythonOperator(
        task_id='get_watermark',
        python_callable=get_watermark
```

```
    )

    extract = SparkSubmitOperator(
        task_id='extract_incremental',
        application='s3://lakehouse/jobs/extract_incremental.py',
        application_args=[
            '--watermark', '{{ ti.xcom_pull(task_ids="get_watermark", key="watermark") }}'
        ]
    )

    update_wm = PythonOperator(
        task_id='update_watermark',
        python_callable=update_watermark,
        trigger_rule='all_success'
    )

    get_wm >> extract >> update_wm
```

Backfill contrôlé :

```
from airflow.operators.python import PythonOperator
from datetime import datetime, timedelta

def generate_backfill_dates(start_date, end_date):
    """Génère les dates à retraiter"""
    dates = []
    current = start_date
    while current <= end_date:
        dates.append(current.strftime('%Y-%m-%d'))
        current += timedelta(days=1)
    return dates

with DAG(
    'backfill_transactions',
    schedule_interval=None,  # Déclenché manuellement
    params={
        'start_date': '2024-01-01',
        'end_date': '2024-01-31'
    }
) as dag:

    def run_backfill(**context):
        start = datetime.strptime(context['params']['start_date'], '%Y-%m-%d')
        end = datetime.strptime(context['params']['end_date'], '%Y-%m-%d')
        dates = generate_backfill_dates(start, end)

        for date in dates:
            # Déclenchement du DAG d'ingestion pour chaque date
            trigger_dag_run('ingestion_quotidienne_ventes', date)

    backfill = PythonOperator(
        task_id='run_backfill',
        python_callable=run_backfill
    )
```

Performance et Optimisation

Dimensionnement des Pipelines

Le dimensionnement correct des pipelines d'ingestion équilibre performance, coûts et fiabilité.

Facteurs de dimensionnement :

Composant	Métrique clé	Formule de base
Partitions Kafka	Parallélisme	Volume/heure ÷ 10 Mo/s/partition
Executors Spark	Capacité traitement	Volume batch ÷ 100 Mo/executor/min
Tasks Flink	Parallélisme streaming	Partitions Kafka × 1-2
Workers Connect	Connecteurs	1 worker par 3-5 connecteurs

Exemple de calcul :

```
Scénario: 10 To/jour, latence cible 15 minutes

Volume par batch (15 min): 10 To ÷ 96 = ~104 Go
Débit requis: 104 Go ÷ 15 min = ~7 Go/min = ~115 Mo/s

Partitions Kafka: 115 Mo/s ÷ 10 Mo/s = 12 partitions (arrondi à 16)
Executors Spark: 104 Go ÷ (100 Mo × 15) = 70 executors
Mémoire totale: 70 × 8 Go = 560 Go

Configuration recommandée:
- Kafka: 16 partitions, 3 brokers, rétention 7 jours
- Spark: 70 executors × 4 cores × 8 Go RAM
- Iceberg: fichiers 512 Mo, compaction quotidienne
```

Optimisation des Écritures Iceberg

Configuration pour haute performance :

```
# Propriétés de table optimisées pour l'ingestion
spark.sql("""
    ALTER TABLE lakehouse.bronze.transactions SET TBLPROPERTIES (
        'write.target-file-size-bytes' = '536870912',
        'write.distribution-mode' = 'hash',
        'write.parquet.compression-codec' = 'zstd',
        'write.parquet.compression-level' = '3',
        'write.metadata.compression-codec' = 'gzip',
        'commit.retry.num-retries' = '10',
        'commit.retry.min-wait-ms' = '100',
        'commit.retry.max-wait-ms' = '60000'
    )
""")
```

Distribution des données :

```
# Distribution par hash pour parallélisme optimal
df.writeTo("lakehouse.bronze.transactions") \
    .option("write.distribution-mode", "hash") \
    .option("write.fanout.enabled", "true") \
    .append()

# Distribution par range pour données ordonnées
df.sortWithinPartitions("timestamp_evenement") \
    .writeTo("lakehouse.bronze.evenements") \
    .option("write.distribution-mode", "range") \
    .append()
```

Performance

Le mode `fanout` permet à chaque tâche Spark d'écrire vers toutes les partitions Iceberg simultanément, améliorant le parallélisme au prix d'une utilisation mémoire accrue. Activez-le pour les pipelines avec de nombreuses partitions cibles.

Monitoring et Observabilité

Métriques clés à surveiller :

Métrique	Seuil d'alerte	Action corrective
Lag consommateur Kafka	> 5 min	Augmenter parallélisme
Durée de commit Iceberg	> 60s	Réduire taille des batches
Fichiers par commit	> 1000	Activer compaction
Taux d'erreurs	> 1%	Investigation qualité données
Utilisation mémoire	> 80%	Augmenter ressources

Dashboard Grafana pour ingestion :

```
# Configuration Prometheus pour métriques Spark
- job_name: 'spark-ingestion'
  static_configs:
    - targets: ['spark-driver:4040']
  metrics_path: /metrics/prometheus

# Alertes critiques
groups:
  - name: ingestion-alerts
    rules:
      - alert: IngestionLagCritique
        expr: kafka_consumer_lag > 300000
        for: 5m
        labels:
          severity: critical
        annotations:
          summary: "Lag d'ingestion critique"
```

```
      - alert: IcebergCommitLent
        expr: iceberg_commit_duration_seconds > 120
        for: 3m
        labels:
          severity: warning
        annotations:
          summary: "Commits Iceberg anormalement lents"
```

Études de Cas Canadiennes

Secteur Télécommunications : Opérateur National

> **Étude de cas : Grand opérateur de télécommunications canadien**
>
> *Secteur* : Télécommunications
>
> *Défi* : Ingérer 500 millions d'événements réseau quotidiens (CDR, logs réseau, métriques QoS) avec une latence maximale de 5 minutes pour alimenter les tableaux de bord opérationnels et les systèmes de détection d'anomalies.
>
> *Solution* : Architecture Streaming Lakehouse avec Kafka (80 partitions), Apache Flink pour le traitement temps réel, et Iceberg sur ADLS Gen2. CDC Debezium pour la synchronisation des données clients depuis les systèmes CRM et de facturation.
>
> *Architecture* :
>
> ```
> Sources réseau → Kafka (500M evt/jour) → Flink → Iceberg Bronze
>                                                    ↓
>                                  Flink (agrégation) → Iceberg Silver
>                                                    ↓
>                                      Dremio → Tableaux de bord
> ```
>
> *Résultats* :
>
> - Latence médiane de 2 minutes (P99 < 5 min)
> - Réduction de 80% du temps de détection des anomalies réseau
> - Économies de 3M$/an par rapport à la solution propriétaire précédente
> - Conformité maintenue avec les exigences du CRTC

Secteur Commerce de Détail : Chaîne Nationale

> **Étude de cas : Chaîne de magasins pancanadienne**
>
> *Secteur* : Commerce de détail

Défi : Unifier les données de 400 magasins, du commerce électronique et des programmes de fidélité pour créer une vue client 360° alimentant la personnalisation en temps réel.

Solution : Ingestion hybride combinant CDC (Debezium) pour les systèmes transactionnels POS, API webhooks pour le commerce électronique, et batch pour les fichiers partenaires. Kafka comme bus central, Spark Structured Streaming pour le micro-batch (5 min), et Iceberg sur S3.

Résultats :

- Vue client unifiée avec latence < 10 minutes
- Augmentation de 15% du taux de conversion des recommandations
- Réduction de 60% du temps de préparation des campagnes marketing
- Capacité d'analyse cross-canal impossible auparavant

Secteur Public : Agence Gouvernementale

Étude de cas : Agence fédérale de statistiques

Secteur : Gouvernement fédéral

Défi : Moderniser l'infrastructure d'ingestion pour le recensement et les enquêtes continues, avec des exigences strictes de sécurité (Protégé B) et de résidence des données au Canada.

Solution : Déploiement sur site avec MinIO pour le stockage, Kafka sur Kubernetes pour le bus événementiel, et Spark pour l'ingestion. Toute l'infrastructure hébergée dans les centres de données gouvernementaux certifiés.

Particularités :

- Chiffrement de bout en bout avec clés gérées par l'agence
- Audit complet de toutes les opérations d'ingestion
- Isolation réseau stricte entre environnements

Résultats :

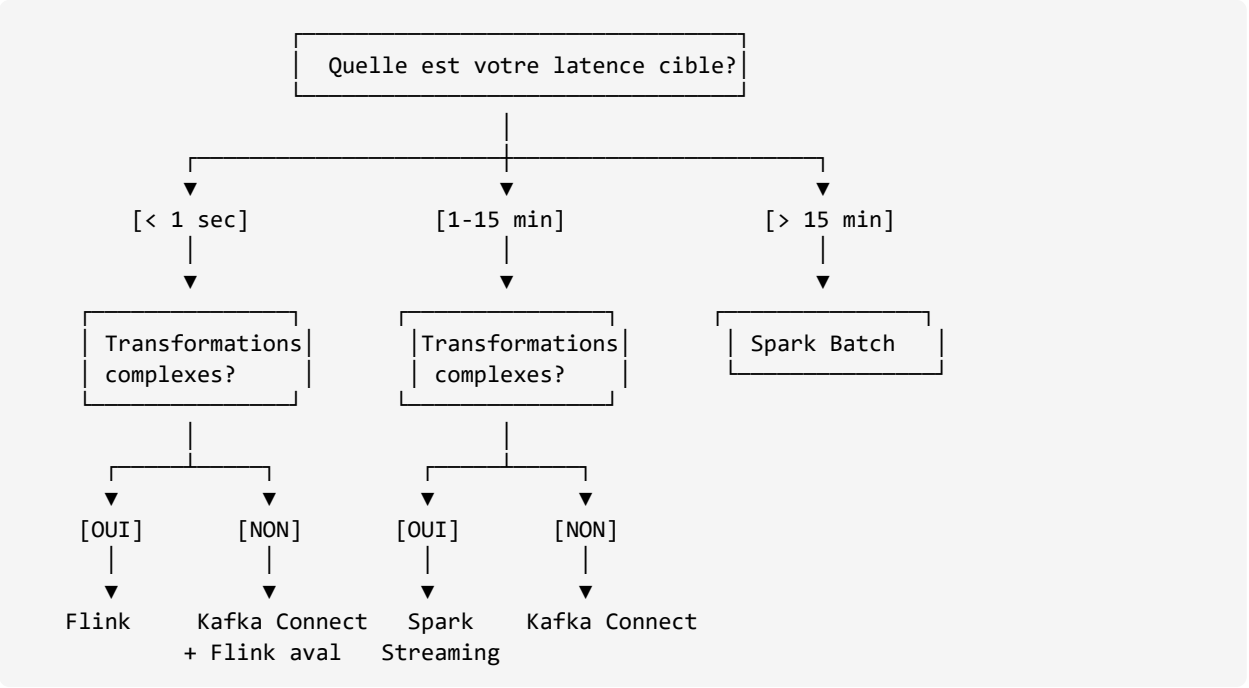
- Conformité Protégé B validée par audit indépendant
- Réduction de 70% du temps de traitement des enquêtes
- Capacité d'intégrer de nouvelles sources en semaines plutôt qu'en mois

Matrice de Décision Technologique

Choix du Moteur d'Ingestion

Critère	Kafka Connect	Spark Streaming	Apache Flink
Latence	1-5 min	1-15 min	< 1 sec
Complexité transformation	Faible	Élevée	Élevée
Exactly-once	Oui (avec config)	Oui	Oui
Courbe apprentissage	Faible	Moyenne	Élevée
Opérations	Simple	Modéré	Complexe
Cas d'usage	CDC direct	ETL batch/micro	Temps réel

Arbre de Décision



Conclusion

L'architecture de la couche d'ingestion détermine la capacité de votre Data Lakehouse à transformer des données brutes dispersées en actifs analytiques exploitables. Une conception rigoureuse de cette couche — combinant les bons patterns d'ingestion, les technologies appropriées et des pratiques opérationnelles solides — constitue le fondement sur lequel repose toute votre stratégie de données.

Apache Iceberg transforme la manière dont nous concevons l'ingestion en unifiant les paradigmes batch et streaming au sein d'une architecture cohérente. Les commits atomiques, l'évolution de schéma et le

support des écritures concurrentes éliminent les compromis historiques entre fraîcheur des données et fiabilité des pipelines.

L'écosystème technologique disponible — Kafka pour le découplage, Debezium pour le CDC, Spark et Flink pour le traitement, Airflow pour l'orchestration — offre des composantes matures et interopérables pour construire des pipelines de classe production. Le choix entre ces technologies dépend de vos exigences spécifiques de latence, de complexité de transformation et de capacités opérationnelles.

Les études de cas canadiennes présentées démontrent que ces architectures s'appliquent avec succès dans des contextes variés — télécommunications, commerce de détail, secteur public — tout en respectant les contraintes réglementaires locales. La résidence des données au Canada, les certifications Protégé B et les exigences sectorielles spécifiques sont compatibles avec une architecture Lakehouse moderne.

Le chapitre suivant explore la couche de catalogue, composante essentielle qui organise et gouverne les métadonnées de votre Lakehouse, permettant la découverte, l'accès contrôlé et l'évolution des tables Iceberg à l'échelle de l'entreprise.

Résumé

Patterns d'ingestion :

- Batch pour les volumes massifs avec latence tolérante (heures)
- Micro-batch pour l'équilibre latence/efficacité (minutes)
- Streaming pour les exigences temps réel (secondes)
- Le Streaming Lakehouse unifie ces patterns avec Iceberg

Technologies clés :

- Apache Kafka comme bus événementiel central
- Debezium pour le CDC depuis les bases relationnelles
- Spark Structured Streaming pour le micro-batch
- Apache Flink pour le streaming temps réel
- Kafka Connect pour l'ingestion directe simple

Stratégies de chargement :

- Append pour les données nouvelles sans mise à jour
- MERGE INTO pour les upserts CDC
- Gestion des suppressions : physique, logique ou historisée

Qualité des données :

- Validation en ligne avec Great Expectations
- Déduplication intégrée aux pipelines
- Architecture de quarantaine pour les données invalides

Optimisations :

- Distribution hash pour parallélisme optimal
- Dimensionnement fichiers 256-512 Mo
- Configuration des retries pour commits robustes
- Monitoring Kafka lag et durée commits

Orchestration :

- Airflow pour les pipelines complexes
- Gestion d'état pour l'ingestion incrémentale
- Backfill contrôlé pour les reprises historiques

Ce chapitre établit les fondations de l'alimentation de votre Lakehouse. Le chapitre suivant, « Configuration du Catalogue Iceberg », détaille comment organiser, gouverner et exposer les métadonnées de vos tables pour une exploitation à l'échelle de l'entreprise.

Chapitre IV.7 - Implémentation de la Couche de Catalogue

Introduction

Le catalogue constitue le cerveau de votre Data Lakehouse Apache Iceberg. Tandis que la couche de stockage héberge physiquement les données et que la couche d'ingestion alimente le système, le catalogue orchestre l'ensemble en maintenant la cartographie complète de vos actifs de données : où se trouvent les tables, comment elles sont structurées, quelles versions existent et qui peut y accéder. Sans catalogue robuste, votre Lakehouse n'est qu'une collection de fichiers Parquet dispersés dans le stockage objet — techniquement accessibles, mais pratiquement inexploitables à l'échelle de l'entreprise.

Apache Iceberg introduit une architecture de catalogue distinctive qui sépare clairement les responsabilités. Le catalogue Iceberg ne stocke pas les métadonnées détaillées des tables ; il maintient plutôt un registre de pointeurs vers les fichiers de métadonnées résidant dans le stockage. Cette indirection élégante permet une flexibilité remarquable : plusieurs moteurs de requête peuvent accéder simultanément aux mêmes tables, les métadonnées évoluent de manière atomique et la migration entre catalogues devient une opération de reconfiguration plutôt qu'un déplacement massif de données.

Ce chapitre vous guide dans la sélection, l'implémentation et l'opération d'une couche de catalogue adaptée à votre contexte. Nous examinerons les différentes implémentations disponibles — du traditionnel Hive Metastore au moderne REST Catalog, en passant par les solutions gérées des fournisseurs infonuagiques —, détaillerons les configurations de production et explorerons les stratégies de fédération pour les architectures multi-environnements.

L'enjeu dépasse la simple gestion technique des métadonnées. Le catalogue devient le point de contrôle central pour la gouvernance des données, la sécurité d'accès et la découverte des actifs. Une implémentation réussie transforme votre Lakehouse d'une infrastructure technique en une plateforme de données gouvernée, où chaque table est documentée, chaque accès est contrôlé et chaque évolution est tracée.

Fondamentaux du Catalogue Iceberg

Rôle et Responsabilités

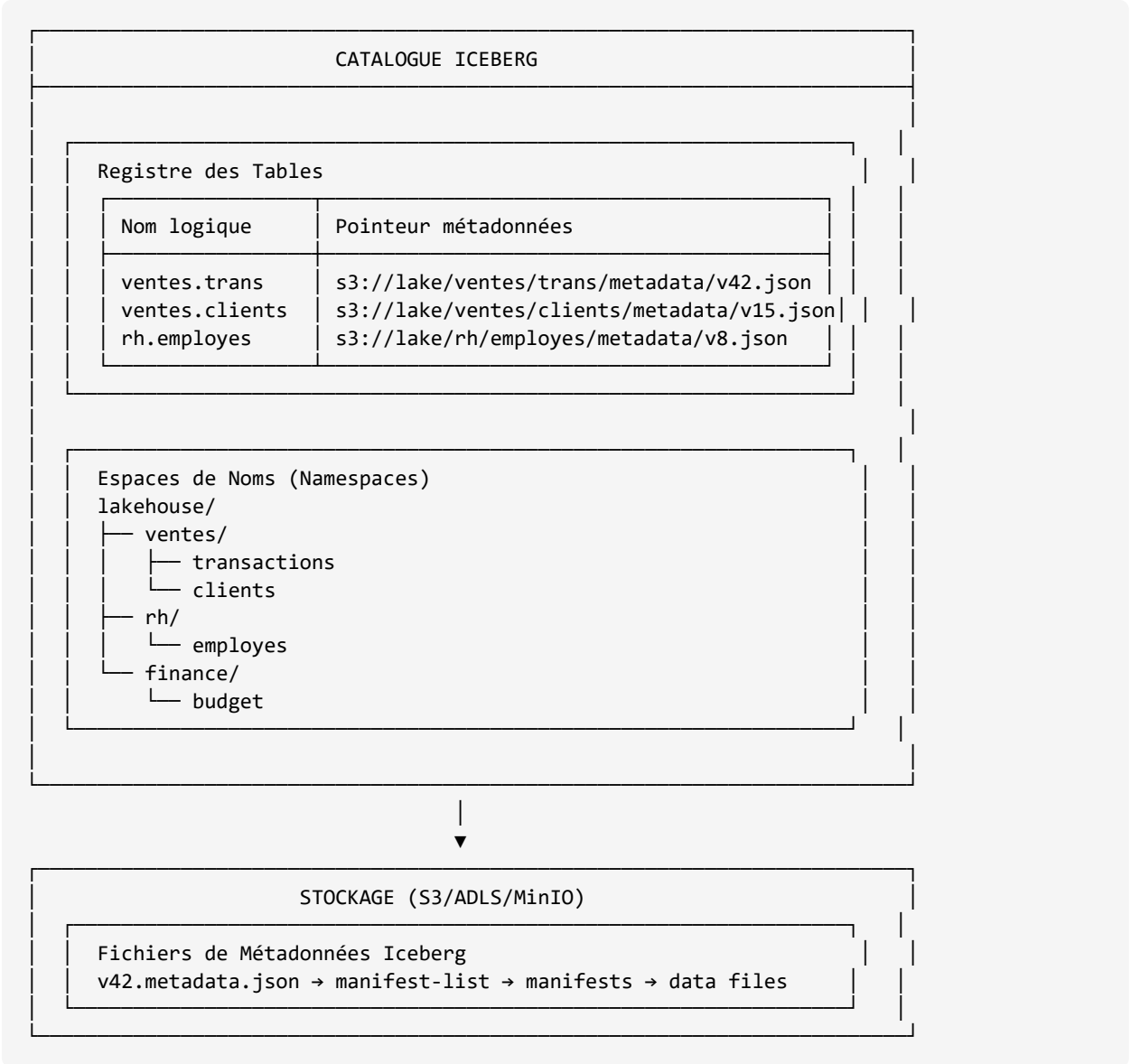
Le catalogue Iceberg assume des responsabilités précises et délimitées dans l'architecture globale du Lakehouse. Comprendre cette délimitation est essentiel pour concevoir une implémentation efficace.

Registre des tables : Le catalogue maintient la correspondance entre les noms logiques des tables (par exemple, `lakehouse.ventes.transactions`) et l'emplacement physique de leurs métadonnées dans le stockage. Cette indirection permet de renommer ou déplacer des tables sans modifier les fichiers sous-jacents.

Gestion des espaces de noms : Les catalogues Iceberg supportent une hiérarchie d'espaces de noms (namespaces) permettant d'organiser logiquement les tables. Cette structure reflète typiquement l'organisation métier : domaines, départements ou environnements.

Contrôle de concurrence : Lors des commits de modifications, le catalogue garantit l'atomicité des mises à jour du pointeur de métadonnées. Ce mécanisme prévient les conditions de course entre écrivains concurrents.

Découverte des métadonnées : Le catalogue expose les informations nécessaires aux moteurs de requête pour localiser et interroger les tables : schéma, partitionnement, statistiques et emplacement des fichiers de manifeste.



Architecture Découplée

L'architecture Iceberg découple intentionnellement le catalogue des métadonnées détaillées. Le catalogue ne contient qu'un pointeur vers le fichier `metadata.json` courant ; toute la richesse des métadonnées — schéma, partitionnement, snapshots, manifestes — réside dans les fichiers du stockage objet.

Cette séparation offre plusieurs avantages :

Portabilité : Les métadonnées complètes voyagent avec les données. Migrer vers un nouveau catalogue nécessite uniquement de réenregistrer les tables avec leurs pointeurs existants.

Scalabilité : Le catalogue reste léger puisqu'il ne stocke que des références. Les métadonnées volumineuses (manifestes pour des millions de fichiers) résident dans le stockage objet élastique.

Cohérence : Les mises à jour atomiques du pointeur garantissent que les lecteurs voient toujours un état cohérent, même pendant les écritures concurrentes.

Interopérabilité : Plusieurs moteurs de requête accèdent aux mêmes métadonnées sans dépendre d'un format de catalogue propriétaire.

Interface Catalog d'Iceberg

Apache Iceberg définit une interface Java standardisée que toutes les implémentations de catalogue doivent respecter :

```
public interface Catalog {
    // Gestion des espaces de noms
    List<Namespace> listNamespaces();
    List<Namespace> listNamespaces(Namespace namespace);
    Map<String, String> loadNamespaceMetadata(Namespace namespace);
    boolean createNamespace(Namespace namespace, Map<String, String> metadata);
    boolean dropNamespace(Namespace namespace);

    // Gestion des tables
    List<TableIdentifier> listTables(Namespace namespace);
    Table loadTable(TableIdentifier identifier);
    Table createTable(TableIdentifier identifier, Schema schema,
                      PartitionSpec spec, Map<String, String> properties);
    boolean dropTable(TableIdentifier identifier, boolean purge);
    void renameTable(TableIdentifier from, TableIdentifier to);

    // Transactions
    Transaction newCreateTableTransaction(TableIdentifier identifier,
                                          Schema schema, PartitionSpec spec);
    Transaction newReplaceTableTransaction(TableIdentifier identifier,
                                           Schema schema, PartitionSpec spec);
}
```

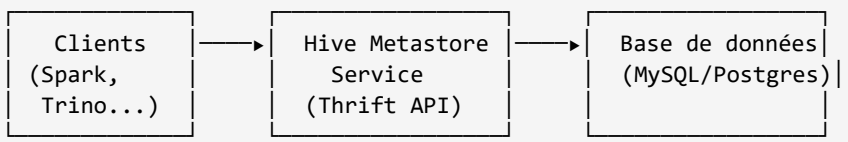
Cette interface standardisée garantit que le code applicatif reste indépendant de l'implémentation de catalogue choisie. Vous pouvez développer avec un catalogue de test local et déployer en production avec un catalogue d'entreprise sans modification de code.

Panorama des Implémentations de Catalogue

Hive Metastore

Le Hive Metastore (HMS) représente l'implémentation historique, héritée de l'écosystème Hadoop. Malgré son ancienneté, il demeure largement déployé et offre une compatibilité étendue avec les outils existants.

Architecture :



Avantages :

- Compatibilité avec l'écosystème Hadoop existant
- Maturité et stabilité éprouvées
- Support natif dans Spark, Trino, Hive
- Documentation et expertise abondantes

Limitations :

- Performance limitée pour les opérations à haute fréquence
- Modèle de sécurité basique
- Complexité opérationnelle (service + base de données)
- Pas de support natif pour les fonctionnalités Iceberg avancées

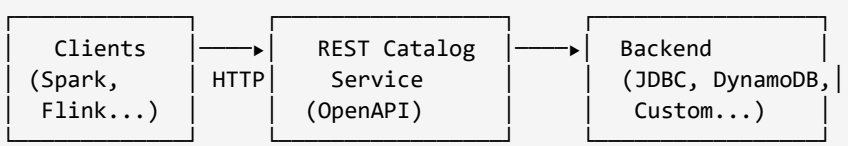
Configuration Spark avec Hive Metastore :

```
spark = SparkSession.builder \
    .appName("IcebergHiveMetastore") \
    .config("spark.sql.catalog.lakehouse", "org.apache.iceberg.spark.SparkCatalog") \
    .config("spark.sql.catalog.lakehouse.type", "hive") \
    .config("spark.sql.catalog.lakehouse.uri", "thrift://hive-metastore:9083") \
    .config("spark.sql.catalog.lakehouse.warehouse", "s3://entreprise-lakehouse/
warehouse") \
    .config("spark.hadoop.hive.metastore.uris", "thrift://hive-metastore:9083") \
    .enableHiveSupport() \
    .getOrCreate()
```

REST Catalog

Le REST Catalog représente l'évolution moderne de l'architecture de catalogue Iceberg. Défini par la spécification OpenAPI d'Iceberg, il standardise les interactions via une API HTTP/JSON, facilitant l'interopérabilité et les déploiements distribués.

Architecture :



Avantages :

- Standard ouvert et bien documenté
- Indépendant du langage et de la plateforme
- Support natif des fonctionnalités Iceberg (views, transactions)
- Flexibilité du backend de stockage
- Facilité d'intégration avec les systèmes d'authentification modernes

Limitations :

- Écosystème moins mature que Hive Metastore
- Nécessite le déploiement d'un service dédié
- Moins d'implémentations de production disponibles

Spécification REST Catalog :

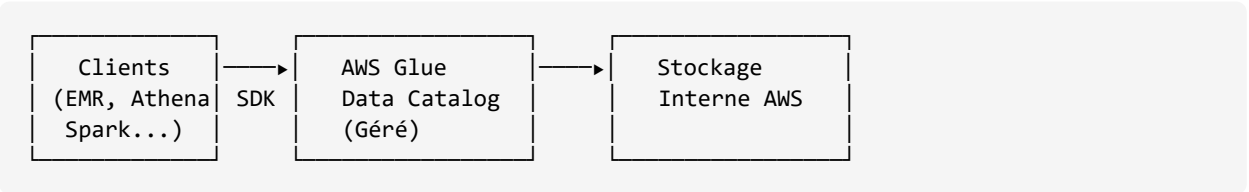
L'API REST Catalog définit des endpoints standardisés :

Endpoint	Méthode	Description
`/v1/config`	GET	Configuration du catalogue
`/v1/namespaces`	GET, POST	Gestion des espaces de noms
`/v1/namespaces/{ns}`	GET, DELETE	Opérations sur un namespace
`/v1/namespaces/{ns}/tables`	GET, POST	Liste et création de tables
`/v1/namespaces/{ns}/tables/{table}`	GET, DELETE	Opérations sur une table
`/v1/namespaces/{ns}/tables/{table}/metrics`	POST	Métriques de scan

AWS Glue Data Catalog

AWS Glue Data Catalog offre une solution entièrement gérée pour les déploiements AWS. Il élimine la charge opérationnelle du Hive Metastore tout en maintenant la compatibilité avec l'écosystème.

Architecture :



Avantages :

- Aucune infrastructure à gérer
- Intégration native avec les services AWS (Athena, EMR, Redshift Spectrum)
- Haute disponibilité garantie par AWS
- Intégration IAM pour la sécurité
- Découverte automatique via Glue Crawlers

Limitations :

- Verrouillage fournisseur AWS
- Coûts potentiellement élevés à grande échelle
- Limitations sur le nombre d'objets et les requêtes par seconde
- Régions canadiennes limitées (ca-central-1, ca-west-1)

Configuration Spark avec Glue Catalog :

```
spark = SparkSession.builder \
    .appName("IcebergGlueCatalog") \
    .config("spark.sql.catalog.glue_catalog", "org.apache.iceberg.spark.SparkCatalog") \
    .config("spark.sql.catalog.glue_catalog.catalog-impl",
            "org.apache.iceberg.aws.glue.GlueCatalog") \
    .config("spark.sql.catalog.glue_catalog.warehouse",
            "s3://entreprise-lakehouse/warehouse") \
    .config("spark.sql.catalog.glue_catalog.io-impl",
            "org.apache.iceberg.aws.s3.S3FileIO") \
    .config("spark.sql.catalog.glue_catalog.glue.skip-archive", "true") \
    .getOrCreate()
```

Azure Purview et Unity Catalog

Pour les environnements Azure, plusieurs options de catalogue coexistent, avec des positionnements distincts.

Azure Purview : Solution de gouvernance des données offrant découverte, classification et lignage. Purview peut indexer les tables Iceberg mais n'agit pas comme catalogue transactionnel Iceberg natif.

Databricks Unity Catalog : Catalogue unifié pour les environnements Databricks, supportant Iceberg via le format Delta Lake avec interopérabilité UniForm. Unity Catalog offre gouvernance fine, lignage et partage de données.

Configuration avec Unity Catalog :

```
# Dans un environnement Databricks avec Unity Catalog
spark.sql("""
    CREATE CATALOG IF NOT EXISTS lakehouse_iceberg
    USING iceberg
""")

spark.sql("""
    CREATE SCHEMA IF NOT EXISTS lakehouse_iceberg.bronze
    LOCATION 's3://entreprise-lakehouse/bronze'
""")
```

Catalogues Commerciaux

Plusieurs éditeurs proposent des catalogues Iceberg de classe entreprise avec des fonctionnalités avancées.

Tabular (acquis par Databricks) : Catalogue SaaS fondé par les créateurs d'Iceberg, offrant gouvernance, collaboration et optimisation automatique.

Dremio Arctic : Catalogue géré intégré à la plateforme Dremio, avec support Git-like pour le versionnement des tables et branches de données.

Starburst Galaxy : Catalogue cloud de Starburst (contributeur majeur à Trino), unifiant l'accès aux données avec gouvernance intégrée.

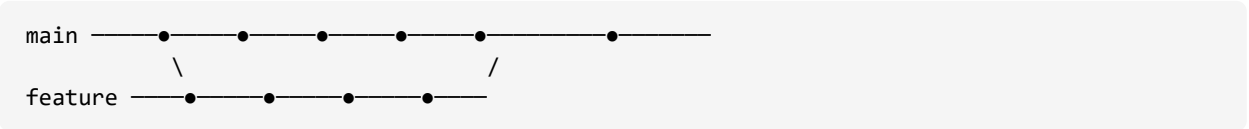
Solution	Modèle	Forces	Considérations
Tabular	SaaS	Expertise Iceberg, optimisation	Acquisition Databricks récente
Dremio Arctic	SaaS/Hybride	Versionnement Git, intégration Dremio	Écosystème Dremio
Starburst Galaxy	SaaS	Fédération, Trino natif	Coût, dépendance
Nessie	Open Source	Git-like, flexible	Maturité, support

Nessie : Le Catalogue Git-like

Project Nessie propose une approche innovante inspirée de Git pour la gestion des catalogues Iceberg. Il permet de créer des branches, de commiter des modifications et de fusionner des changements, apportant les pratiques DevOps au monde des données.

Concepts clés :

- **Branches** : Versions parallèles du catalogue pour développement isolé
- **Commits** : Modifications atomiques avec historique complet
- **Tags** : Points de référence immuables pour les releases
- **Merge** : Fusion de branches avec détection de conflits



Cas d'usage :

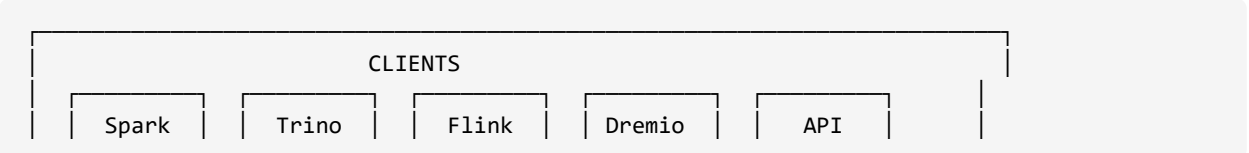
- Environnements de développement isolés
- Tests de migrations de schéma avant déploiement
- Rollback transactionnel multi-tables
- Audit et conformité avec historique complet

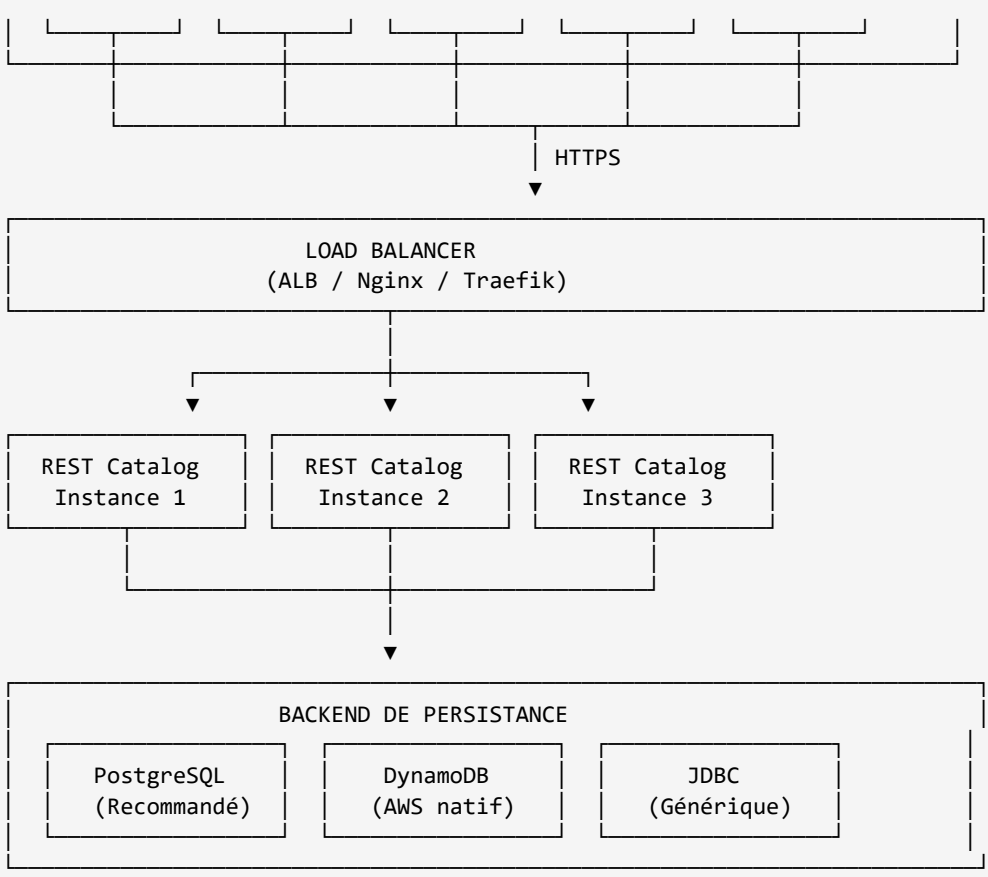
Implémentation du REST Catalog

Architecture de Déploiement

Le REST Catalog nécessite le déploiement d'un service HTTP exposant l'API standardisée Iceberg. Plusieurs implémentations de référence existent.

Architecture de production :





Déploiement avec Polaris (Incubateur Apache)

Apache Polaris (anciennement Snowflake Polaris Catalog, donné à la fondation Apache) représente l'implémentation REST Catalog de référence open source.

Déploiement Kubernetes :

```yaml
# polaris-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: polaris-catalog
  namespace: lakehouse
spec:
  replicas: 3
  selector:
    matchLabels:
      app: polaris-catalog
  template:
    metadata:
      labels:
        app: polaris-catalog
    spec:
      containers:
        - name: polaris
          image: apache/polaris:latest
          ports:
            - containerPort: 8181
```

```
            name: http
          - containerPort: 8182
            name: management
        env:
          - name: POLARIS_PERSISTENCE_TYPE
            value: "eclipse-link"
          - name: POLARIS_PERSISTENCE_ECLIPSELINK_URL
            valueFrom:
              secretKeyRef:
                name: polaris-db-credentials
                key: jdbc-url
          - name: POLARIS_PERSISTENCE_ECLIPSELINK_USER
            valueFrom:
              secretKeyRef:
                name: polaris-db-credentials
                key: username
          - name: POLARIS_PERSISTENCE_ECLIPSELINK_PASSWORD
            valueFrom:
              secretKeyRef:
                name: polaris-db-credentials
                key: password
          - name: POLARIS_DEFAULT_REALM
            value: "entreprise"
          - name: AWS_REGION
            value: "ca-central-1"
        resources:
          requests:
            memory: "2Gi"
            cpu: "1000m"
          limits:
            memory: "4Gi"
            cpu: "2000m"
        livenessProbe:
          httpGet:
            path: /healthcheck
            port: 8182
          initialDelaySeconds: 30
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /healthcheck
            port: 8182
          initialDelaySeconds: 10
          periodSeconds: 5
---
apiVersion: v1
kind: Service
metadata:
  name: polaris-catalog
  namespace: lakehouse
spec:
  selector:
    app: polaris-catalog
  ports:
    - name: http
      port: 8181
      targetPort: 8181
    - name: management
      port: 8182
```

```
      targetPort: 8182
  type: ClusterIP
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: polaris-catalog-ingress
  namespace: lakehouse
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
    cert-manager.io/cluster-issuer: "letsencrypt-prod"
spec:
  tls:
    - hosts:
        - catalog.lakehouse.entreprise.ca
      secretName: polaris-tls
  rules:
    - host: catalog.lakehouse.entreprise.ca
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: polaris-catalog
                port:
                  number: 8181
```

Configuration de la base de données PostgreSQL :

```
# postgresql-polaris.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: polaris-db-init
  namespace: lakehouse
data:
  init.sql: |
    CREATE DATABASE polaris_catalog;
    CREATE USER polaris WITH ENCRYPTED PASSWORD '${POLARIS_DB_PASSWORD}';
    GRANT ALL PRIVILEGES ON DATABASE polaris_catalog TO polaris;

    \c polaris_catalog

    -- Schéma de persistance Polaris
    CREATE SCHEMA IF NOT EXISTS polaris;
    GRANT ALL ON SCHEMA polaris TO polaris;
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: polaris-postgresql
  namespace: lakehouse
spec:
  serviceName: polaris-postgresql
  replicas: 1
  selector:
```

```
    matchLabels:
      app: polaris-postgresql
  template:
    metadata:
      labels:
        app: polaris-postgresql
    spec:
      containers:
        - name: postgresql
          image: postgres:15
          ports:
            - containerPort: 5432
          env:
            - name: POSTGRES_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: polaris-db-credentials
                  key: postgres-password
            - name: PGDATA
              value: /var/lib/postgresql/data/pgdata
          volumeMounts:
            - name: data
              mountPath: /var/lib/postgresql/data
            - name: init-scripts
              mountPath: /docker-entrypoint-initdb.d
          resources:
            requests:
              memory: "1Gi"
              cpu: "500m"
      volumes:
        - name: init-scripts
          configMap:
            name: polaris-db-init
  volumeClaimTemplates:
    - metadata:
        name: data
      spec:
        accessModes: ["ReadWriteOnce"]
        storageClassName: gp3
        resources:
          requests:
            storage: 100Gi
```

Configuration Client

Configuration Spark :

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("IcebergRESTCatalog") \
    .config("spark.sql.extensions",
            "org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions") \
    .config("spark.sql.catalog.lakehouse",
            "org.apache.iceberg.spark.SparkCatalog") \
    .config("spark.sql.catalog.lakehouse.type", "rest") \
    .config("spark.sql.catalog.lakehouse.uri",
```

```
            "https://catalog.lakehouse.entreprise.ca") \
    .config("spark.sql.catalog.lakehouse.warehouse",
            "s3://entreprise-lakehouse/warehouse") \
    .config("spark.sql.catalog.lakehouse.credential",
            "${CATALOG_TOKEN}") \
    .config("spark.sql.catalog.lakehouse.token",
            "${CATALOG_OAUTH_TOKEN}") \
    .config("spark.sql.catalog.lakehouse.io-impl",
            "org.apache.iceberg.aws.s3.S3FileIO") \
    .config("spark.sql.catalog.lakehouse.s3.endpoint",
            "s3.ca-central-1.amazonaws.com") \
    .getOrCreate()
```

Configuration Trino :

```
# etc/catalog/lakehouse.properties
connector.name=iceberg
iceberg.catalog.type=rest
iceberg.rest-catalog.uri=https://catalog.lakehouse.entreprise.ca
iceberg.rest-catalog.warehouse=s3://entreprise-lakehouse/warehouse

# Authentification OAuth2
iceberg.rest-catalog.security=OAUTH2
iceberg.rest-catalog.oauth2.token=file:///etc/trino/secrets/oauth-token

# Configuration S3
iceberg.file-system-type=S3
s3.region=ca-central-1
s3.endpoint=s3.ca-central-1.amazonaws.com
```

Configuration Flink :

```
CREATE CATALOG lakehouse WITH (
    'type' = 'iceberg',
    'catalog-type' = 'rest',
    'uri' = 'https://catalog.lakehouse.entreprise.ca',
    'warehouse' = 's3://entreprise-lakehouse/warehouse',
    'io-impl' = 'org.apache.iceberg.aws.s3.S3FileIO'
);

USE CATALOG lakehouse;
```

Implémentation AWS Glue Catalog

Configuration et Déploiement

AWS Glue Data Catalog ne nécessite aucun déploiement d'infrastructure — le service est entièrement géré. La configuration se concentre sur les permissions IAM et les paramètres de connexion.

Politique IAM pour accès Glue Catalog :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "GlueCatalogAccess",
            "Effect": "Allow",
            "Action": [
                "glue:GetDatabase",
                "glue:GetDatabases",
                "glue:CreateDatabase",
                "glue:UpdateDatabase",
                "glue:DeleteDatabase",
                "glue:GetTable",
                "glue:GetTables",
                "glue:CreateTable",
                "glue:UpdateTable",
                "glue:DeleteTable",
                "glue:GetPartition",
                "glue:GetPartitions",
                "glue:BatchGetPartition",
                "glue:CreatePartition",
                "glue:BatchCreatePartition",
                "glue:UpdatePartition",
                "glue:DeletePartition",
                "glue:BatchDeletePartition"
            ],
            "Resource": [
                "arn:aws:glue:ca-central-1:${AWS_ACCOUNT_ID}:catalog",
                "arn:aws:glue:ca-central-1:${AWS_ACCOUNT_ID}:database/lakehouse_*",
                "arn:aws:glue:ca-central-1:${AWS_ACCOUNT_ID}:table/lakehouse_*/*"
            ]
        },
        {
            "Sid": "S3DataAccess",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:PutObject",
                "s3:DeleteObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::entreprise-lakehouse",
                "arn:aws:s3:::entreprise-lakehouse/*"
            ]
        }
    ]
}
```

Configuration Terraform :

```
# glue-catalog.tf

# Base de données Glue pour le Lakehouse
resource "aws_glue_catalog_database" "lakehouse_bronze" {
  name        = "lakehouse_bronze"
  description = "Couche Bronze du Lakehouse - Données brutes"
```

```
  location_uri = "s3://entreprise-lakehouse/bronze/"

  create_table_default_permission {
    permissions = ["ALL"]

    principal {
      data_lake_principal_identifier = "IAM_ALLOWED_PRINCIPALS"
    }
  }
}

resource "aws_glue_catalog_database" "lakehouse_silver" {
  name        = "lakehouse_silver"
  description = "Couche Silver du Lakehouse - Données nettoyées"

  location_uri = "s3://entreprise-lakehouse/silver/"
}

resource "aws_glue_catalog_database" "lakehouse_gold" {
  name        = "lakehouse_gold"
  description = "Couche Gold du Lakehouse - Données agrégées"

  location_uri = "s3://entreprise-lakehouse/gold/"
}

# Politique de ressources pour le catalogue
resource "aws_glue_resource_policy" "lakehouse_policy" {
  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Sid       = "CrossAccountAccess"
        Effect    = "Allow"
        Principal = {
          AWS = [
            "arn:aws:iam::${var.analytics_account_id}:root"
          ]
        }
        Action = [
          "glue:GetDatabase",
          "glue:GetDatabases",
          "glue:GetTable",
          "glue:GetTables",
          "glue:GetPartition",
          "glue:GetPartitions"
        ]
        Resource = [
          "arn:aws:glue:ca-central-1:
${data.aws_caller_identity.current.account_id}:catalog",
          "arn:aws:glue:ca-central-1:
${data.aws_caller_identity.current.account_id}:database/lakehouse_*",
          "arn:aws:glue:ca-central-1:${data.aws_caller_identity.current.account_id}:table/
lakehouse_*/*"
        ]
      }
    ]
  })
}
```

```
# Configuration Lake Formation pour gouvernance avancée
resource "aws_lakeformation_data_lake_settings" "lakehouse" {
  admins = [
    data.aws_iam_role.data_platform_admin.arn
  ]

  create_database_default_permissions {
    permissions = ["ALL"]
    principal   = "IAM_ALLOWED_PRINCIPALS"
  }

  create_table_default_permissions {
    permissions = ["ALL"]
    principal   = "IAM_ALLOWED_PRINCIPALS"
  }
}
```

Intégration avec Lake Formation

AWS Lake Formation ajoute une couche de gouvernance fine au-dessus de Glue Catalog, permettant un contrôle d'accès au niveau des colonnes et des lignes.

Configuration des permissions Lake Formation :

```
# Permissions au niveau table
resource "aws_lakeformation_permissions" "analysts_read" {
  principal = aws_iam_role.data_analysts.arn

  permissions = ["SELECT"]

  table {
    database_name = aws_glue_catalog_database.lakehouse_gold.name
    name          = "ventes_agregees"
  }
}

# Permissions au niveau colonne
resource "aws_lakeformation_permissions" "analysts_partial" {
  principal = aws_iam_role.data_analysts.arn

  permissions = ["SELECT"]

  table_with_columns {
    database_name = aws_glue_catalog_database.lakehouse_silver.name
    name          = "clients"

    # Exclusion des colonnes sensibles
    excluded_column_names = [
      "numero_assurance_sociale",
      "date_naissance",
      "adresse_complete"
    ]
  }
}

# Filtrage au niveau ligne
resource "aws_lakeformation_data_cells_filter" "region_quebec" {
```

```
  table_data {
    database_name = aws_glue_catalog_database.lakehouse_silver.name
    name          = "transactions"

    table_catalog_id = data.aws_caller_identity.current.account_id
  }

  name = "filter_quebec_only"

  row_filter {
    filter_expression = "province = 'QC'"
  }
}
```

> **Performance**
>
> AWS Glue Catalog impose des limites de débit : 10 requêtes GetTable par seconde par compte. Pour les charges de travail intensives, activez le cache de métadonnées côté client et envisagez de regrouper les opérations. Les requêtes ListTables avec filtrage côté client sont plus efficaces que de multiples GetTable individuels.

Implémentation Nessie

Concepts et Architecture

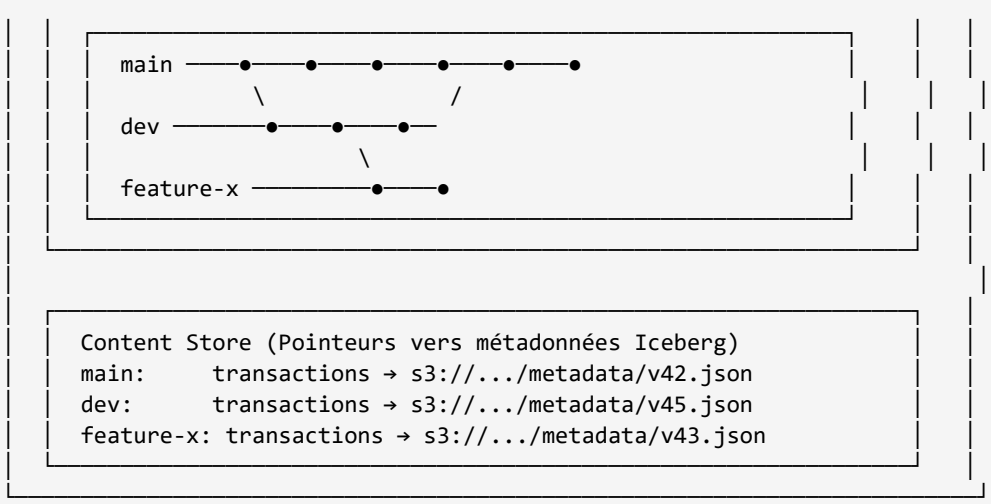
Nessie apporte les concepts de contrôle de version Git au monde des catalogues de données. Cette approche révolutionne la gestion des environnements de développement et de test.

Concepts fondamentaux :

Concept Git	Équivalent Nessie	Usage Lakehouse
Branch	Branch	Environnement isolé (dev, test, prod)
Commit	Commit	Modification atomique multi-tables
Tag	Tag	Version release, point de restauration
Merge	Merge	Promotion dev → prod
Cherry-pick	Transplant	Migration sélective de tables

Architecture Nessie :

```
┌─────────────────────────────────────────────────────────────────┐
│                        NESSIE SERVER                            │
├─────────────────────────────────────────────────────────────────┤
│  ┌──────────────────────────────────────────────────────┐       │
│  │  Version Store                                       │       │
```



Déploiement Nessie

Déploiement Kubernetes avec backend JDBC :

```yaml
# nessie-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nessie-server
  namespace: lakehouse
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nessie
  template:
    metadata:
      labels:
        app: nessie
    spec:
      containers:
        - name: nessie
          image: ghcr.io/projectnessie/nessie:latest
          ports:
            - containerPort: 19120
              name: http
            - containerPort: 9000
              name: management
          env:
            - name: NESSIE_VERSION_STORE_TYPE
              value: "JDBC"
            - name: QUARKUS_DATASOURCE_JDBC_URL
              valueFrom:
                secretKeyRef:
                  name: nessie-db-credentials
                  key: jdbc-url
            - name: QUARKUS_DATASOURCE_USERNAME
              valueFrom:
                secretKeyRef:
                  name: nessie-db-credentials
```

```
                key: username
            - name: QUARKUS_DATASOURCE_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: nessie-db-credentials
                  key: password
            - name: NESSIE_SERVER_DEFAULT_BRANCH
              value: "main"
            - name: NESSIE_SERVER_SEND_STACKTRACE_TO_CLIENT
              value: "false"
          resources:
            requests:
              memory: "1Gi"
              cpu: "500m"
            limits:
              memory: "2Gi"
              cpu: "1000m"
          livenessProbe:
            httpGet:
              path: /q/health/live
              port: 9000
            initialDelaySeconds: 30
          readinessProbe:
            httpGet:
              path: /q/health/ready
              port: 9000
            initialDelaySeconds: 10
---
apiVersion: v1
kind: Service
metadata:
  name: nessie
  namespace: lakehouse
spec:
  selector:
    app: nessie
  ports:
    - name: http
      port: 19120
      targetPort: 19120
```

Flux de Travail Git-like

Configuration Spark avec Nessie :

```
spark = SparkSession.builder \
    .appName("IcebergNessie") \
    .config("spark.sql.catalog.nessie",
            "org.apache.iceberg.spark.SparkCatalog") \
    .config("spark.sql.catalog.nessie.catalog-impl",
            "org.apache.iceberg.nessie.NessieCatalog") \
    .config("spark.sql.catalog.nessie.uri",
            "http://nessie:19120/api/v2") \
    .config("spark.sql.catalog.nessie.ref", "main") \
    .config("spark.sql.catalog.nessie.warehouse",
            "s3://entreprise-lakehouse/warehouse") \
    .config("spark.sql.catalog.nessie.authentication.type", "BEARER") \
```

```
    .config("spark.sql.catalog.nessie.authentication.token",
            "${NESSIE_TOKEN}") \
    .getOrCreate()
```

Opérations de branchement :

```
from pynessie import init as nessie_init
from pynessie.model import Branch, Tag

# Initialisation du client Nessie
client = nessie_init(endpoint="http://nessie:19120/api/v2")

# Création d'une branche de développement
dev_branch = client.create_reference(
    Branch(name="dev-feature-123", hash_=client.get_default_branch().hash_)
)
print(f"Branche créée: {dev_branch.name} à {dev_branch.hash_}")

# Basculement vers la branche de développement dans Spark
spark.conf.set("spark.sql.catalog.nessie.ref", "dev-feature-123")

# Modifications sur la branche dev
spark.sql("""
    ALTER TABLE nessie.bronze.transactions
    ADD COLUMN nouvelle_colonne STRING
""")

# Validation des modifications
client.commit(
    branch="dev-feature-123",
    message="Ajout colonne nouvelle_colonne à transactions",
    author="data-engineer@entreprise.ca"
)

# Merge vers main après validation
client.merge(
    from_ref="dev-feature-123",
    to_ref="main",
    message="Merge feature-123: nouvelle colonne transactions"
)

# Création d'un tag pour la release
client.create_reference(
    Tag(name="release-2024-01-15", hash_=client.get_reference("main").hash_)
)
```

Requêtes sur différentes branches :

```
-- Requête sur la branche principale
SELECT * FROM nessie.bronze.transactions VERSION AS OF 'main';

-- Requête sur une branche de développement
SELECT * FROM nessie.bronze.transactions VERSION AS OF 'dev-feature-123';

-- Requête sur un tag spécifique (point dans le temps)
SELECT * FROM nessie.bronze.transactions VERSION AS OF 'release-2024-01-15';
```

```
-- Time travel combiné avec branching
SELECT * FROM nessie.bronze.transactions
VERSION AS OF 'main'
FOR SYSTEM_TIME AS OF '2024-01-10 10:00:00';
```

Migration

De : Environnements dev/test/prod avec copies de données

Vers : Branches Nessie partageant les mêmes données sous-jacentes

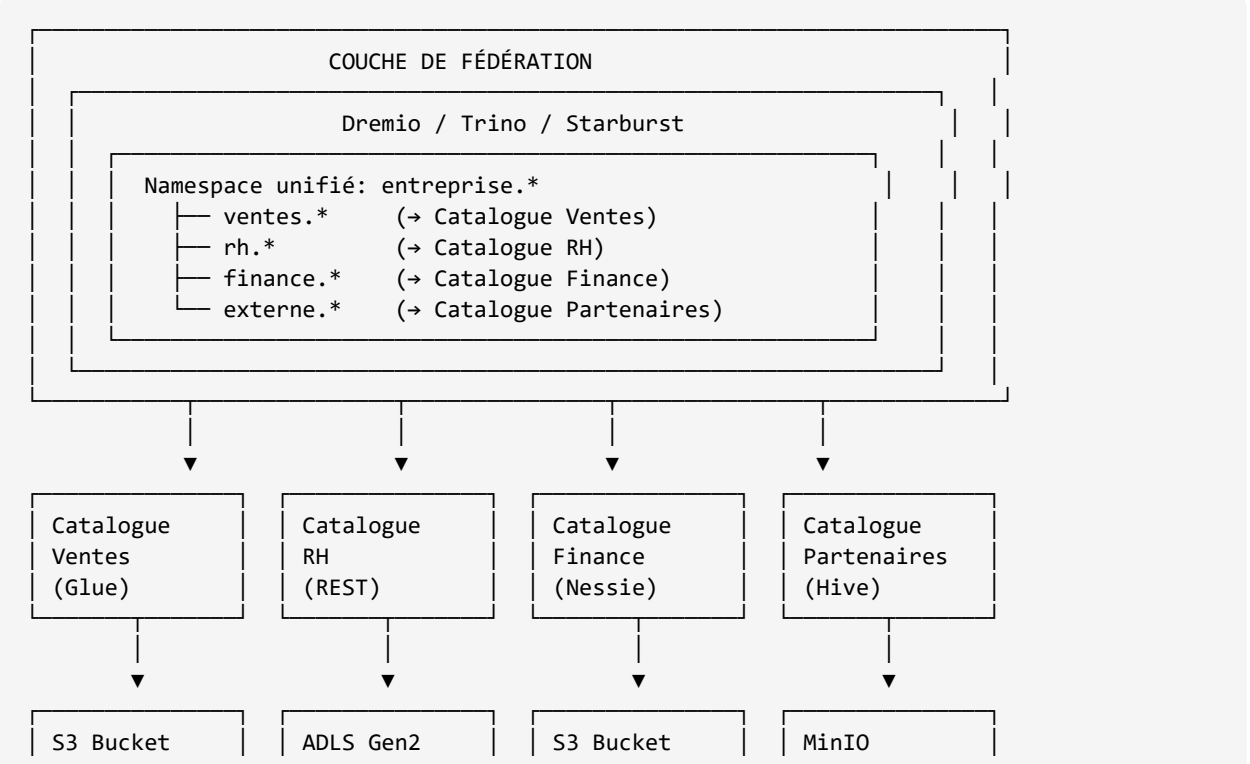
Stratégie : Les branches Nessie ne dupliquent pas les données — elles créent des vues isolées sur les mêmes fichiers. Seules les modifications divergentes créent de nouveaux fichiers. Cette approche réduit drastiquement les coûts de stockage et accélère la création d'environnements.

Fédération Multi-Catalogue

Architecture de Fédération

Les grandes organisations opèrent souvent plusieurs catalogues : un par département, par région ou par niveau de sensibilité des données. La fédération permet d'unifier l'accès tout en préservant l'autonomie de chaque domaine.

Architecture fédérée :



```
┌──────────────┐  ┌──────────────┐  ┌──────────────┐  ┌──────────────┐
│ ca-central-1 │  │ Canada East  │  │ ca-central-1 │  │ On-premise   │
└──────────────┘  └──────────────┘  └──────────────┘  └──────────────┘
```

Configuration Trino Multi-Catalogue

```
# etc/catalog/ventes.properties
connector.name=iceberg
iceberg.catalog.type=glue
iceberg.glue.region=ca-central-1
iceberg.file-system-type=S3

# etc/catalog/rh.properties
connector.name=iceberg
iceberg.catalog.type=rest
iceberg.rest-catalog.uri=https://catalog-rh.entreprise.ca
iceberg.file-system-type=AZURE

# etc/catalog/finance.properties
connector.name=iceberg
iceberg.catalog.type=nessie
iceberg.nessie-catalog.uri=http://nessie-finance:19120/api/v2
iceberg.nessie-catalog.ref=main
iceberg.file-system-type=S3

# etc/catalog/partenaires.properties
connector.name=iceberg
iceberg.catalog.type=hive_metastore
hive.metastore.uri=thrift://hive-partenaires:9083
iceberg.file-system-type=S3
```

Requêtes fédérées :

```
-- Jointure entre catalogues
SELECT
    v.transaction_id,
    v.montant,
    c.nom_client,
    e.nom_employe as vendeur
FROM ventes.gold.transactions v
JOIN rh.silver.employes e ON v.employe_id = e.employe_id
JOIN partenaires.bronze.clients c ON v.client_id = c.client_id
WHERE v.date_transaction >= DATE '2024-01-01';

-- Agrégation cross-catalogue
SELECT
    f.centre_cout,
    SUM(v.montant) as ventes_totales,
    COUNT(DISTINCT v.employe_id) as nb_vendeurs
FROM ventes.gold.transactions v
JOIN finance.silver.centres_cout f ON v.departement_id = f.departement_id
GROUP BY f.centre_cout;
```

Dremio comme Couche de Fédération

Dremio excelle dans la fédération de sources hétérogènes avec une couche sémantique unifiée.

Configuration des sources Dremio :

```
-- Source Iceberg sur S3 avec Glue
ALTER SOURCE "Lakehouse Ventes" ADD
LOCATION 's3://entreprise-ventes/warehouse'
FORMAT iceberg
CATALOG glue
REGION 'ca-central-1';

-- Source Iceberg sur ADLS avec REST Catalog
ALTER SOURCE "Lakehouse RH" ADD
LOCATION 'abfss://lakehouse@entrepriserh.dfs.core.windows.net/warehouse'
FORMAT iceberg
CATALOG rest
URI 'https://catalog-rh.entreprise.ca';

-- Création d'un espace virtuel unifié
CREATE VDS "Espace Unifié"."Ventes avec Vendeurs" AS
SELECT
    t.*,
    e.nom as vendeur_nom,
    e.departement
FROM "Lakehouse Ventes".gold.transactions t
LEFT JOIN "Lakehouse RH".silver.employes e
    ON t.employe_id = e.employe_id;
```

Sécurité et Gouvernance du Catalogue

Modèles d'Authentification

OAuth 2.0 / OIDC :

L'authentification moderne repose sur OAuth 2.0 et OpenID Connect, permettant l'intégration avec les fournisseurs d'identité d'entreprise (Azure AD, Okta, Keycloak).

```
# Configuration Polaris avec OAuth2
polaris:
  authentication:
    type: oauth2
    oauth2:
      issuer-url: https://login.microsoftonline.com/${TENANT_ID}/v2.0
      client-id: ${POLARIS_CLIENT_ID}
      client-secret: ${POLARIS_CLIENT_SECRET}
      scopes:
        - openid
        - profile
        - api://polaris/.default
      audience: api://polaris
```

Configuration client Spark avec OAuth :

```
spark = SparkSession.builder \
    .config("spark.sql.catalog.lakehouse.credential",
            "${OAUTH_ACCESS_TOKEN}") \
    .config("spark.sql.catalog.lakehouse.oauth2-server-uri",
            "https://login.microsoftonline.com/${TENANT_ID}/oauth2/v2.0/token") \
    .config("spark.sql.catalog.lakehouse.oauth2.client-id",
            "${CLIENT_ID}") \
    .config("spark.sql.catalog.lakehouse.oauth2.client-secret",
            "${CLIENT_SECRET}") \
    .config("spark.sql.catalog.lakehouse.oauth2.scope",
            "api://polaris/.default") \
    .getOrCreate()
```

Contrôle d'Accès Granulaire

Modèle RBAC (Role-Based Access Control) :

```
-- Création des rôles dans Polaris
CREATE ROLE data_analyst;
CREATE ROLE data_engineer;
CREATE ROLE data_admin;

-- Attribution des privilèges
GRANT USAGE ON CATALOG lakehouse TO ROLE data_analyst;
GRANT SELECT ON NAMESPACE lakehouse.gold TO ROLE data_analyst;

GRANT USAGE ON CATALOG lakehouse TO ROLE data_engineer;
GRANT SELECT, INSERT, UPDATE, DELETE ON NAMESPACE lakehouse.bronze TO ROLE data_engineer;
GRANT SELECT, INSERT, UPDATE, DELETE ON NAMESPACE lakehouse.silver TO ROLE data_engineer;
GRANT SELECT ON NAMESPACE lakehouse.gold TO ROLE data_engineer;

GRANT ALL PRIVILEGES ON CATALOG lakehouse TO ROLE data_admin;

-- Attribution des rôles aux utilisateurs/groupes
GRANT ROLE data_analyst TO GROUP 'Analystes-BI@entreprise.ca';
GRANT ROLE data_engineer TO GROUP 'Data-Engineering@entreprise.ca';
GRANT ROLE data_admin TO USER 'admin@entreprise.ca';
```

Masquage des données sensibles :

```
-- Politique de masquage pour données PII
CREATE MASKING POLICY mask_courriel AS (val STRING)
RETURNS STRING ->
    CASE
        WHEN CURRENT_ROLE() IN ('data_admin', 'data_engineer') THEN val
        ELSE REGEXP_REPLACE(val, '(.{2}).*@', '$1***@')
    END;

-- Application de la politique
ALTER TABLE lakehouse.silver.clients
ALTER COLUMN courriel SET MASKING POLICY mask_courriel;

-- Politique de masquage pour NAS
CREATE MASKING POLICY mask_nas AS (val STRING)
RETURNS STRING ->
```

```
    CASE
        WHEN CURRENT_ROLE() = 'data_admin' THEN val
        ELSE '***-***-' || RIGHT(val, 3)
    END;
```

Audit et Traçabilité

Configuration de l'audit :

```yaml
# Configuration Polaris pour audit
polaris:
  audit:
    enabled: true
    log-level: INFO
    include-request-body: false
    include-response-body: false

    # Export vers CloudWatch (AWS)
    cloudwatch:
      enabled: true
      log-group: /polaris/audit
      region: ca-central-1

    # Export vers fichier
    file:
      enabled: true
      path: /var/log/polaris/audit.log
      rotation:
        max-size: 100MB
        max-files: 30
```

Structure des événements d'audit :

```json
{
  "timestamp": "2024-01-15T14:30:22.123Z",
  "event_type": "TABLE_READ",
  "catalog": "lakehouse",
  "namespace": "gold",
  "table": "transactions",
  "principal": {
    "type": "USER",
    "name": "analyst@entreprise.ca",
    "roles": ["data_analyst"]
  },
  "action": "SELECT",
  "result": "SUCCESS",
  "client": {
    "application": "Trino",
    "version": "435",
    "ip_address": "10.0.1.45"
  },
  "query_id": "20240115_143022_00042_abcde",
  "rows_returned": 15423,
  "execution_time_ms": 2340
}
```

Requêtes d'audit :

```
-- Analyse des accès par utilisateur (dernières 24h)
SELECT
    principal_name,
    COUNT(*) as nb_requetes,
    COUNT(DISTINCT table_name) as tables_accedees,
    SUM(rows_returned) as lignes_totales
FROM audit_logs
WHERE timestamp >= CURRENT_TIMESTAMP - INTERVAL '24' HOUR
GROUP BY principal_name
ORDER BY nb_requetes DESC;

-- Détection d'accès anormaux
SELECT
    principal_name,
    table_name,
    COUNT(*) as nb_acces,
    AVG(rows_returned) as moy_lignes
FROM audit_logs
WHERE timestamp >= CURRENT_TIMESTAMP - INTERVAL '7' DAY
GROUP BY principal_name, table_name
HAVING COUNT(*) > 100
   AND AVG(rows_returned) > 100000;
```

Migration entre Catalogues

Stratégies de Migration

La migration entre catalogues constitue une opération délicate nécessitant une planification rigoureuse. Trois stratégies principales s'offrent à vous :

Migration par réenregistrement : Les données restent en place ; seuls les pointeurs sont recréés dans le nouveau catalogue. C'est l'approche la plus rapide et la moins risquée.

Migration avec déplacement : Les données sont copiées vers un nouvel emplacement avec enregistrement dans le nouveau catalogue. Nécessaire lors de changement de fournisseur de stockage.

Migration progressive : Les tables sont migrées individuellement avec une période de coexistence des deux catalogues.

Migration Hive Metastore vers REST Catalog

Phase 1 : Inventaire et préparation

```
from pyspark.sql import SparkSession

# Connexion au Hive Metastore existant
spark_hive = SparkSession.builder \
    .config("spark.sql.catalog.hive_catalog", "org.apache.iceberg.spark.SparkCatalog") \
    .config("spark.sql.catalog.hive_catalog.type", "hive") \
    .config("spark.sql.catalog.hive_catalog.uri", "thrift://hive-metastore:9083") \
```

```
    .enableHiveSupport() \
    .getOrCreate()

# Inventaire des tables Iceberg
tables_iceberg = spark_hive.sql("""
    SELECT
        database_name,
        table_name,
        table_location,
        table_properties
    FROM hive_catalog.information_schema.tables
    WHERE table_type = 'ICEBERG'
""").collect()

# Export de l'inventaire
inventaire = []
for table in tables_iceberg:
    metadata_location = spark_hive.sql(f"""
        SELECT current_snapshot_id, metadata_location
        FROM hive_catalog.{table.database_name}.{table.table_name}.metadata_log_entries
        ORDER BY timestamp DESC LIMIT 1
    """).collect()[0]

    inventaire.append({
        'database': table.database_name,
        'table': table.table_name,
        'location': table.table_location,
        'metadata_location': metadata_location.metadata_location
    })

print(f"Tables à migrer: {len(inventaire)}")
```

Phase 2 : Création des namespaces dans le nouveau catalogue

```
import requests

REST_CATALOG_URL = "https://catalog.lakehouse.entreprise.ca"
HEADERS = {"Authorization": f"Bearer {OAUTH_TOKEN}", "Content-Type": "application/json"}

# Extraction des namespaces uniques
namespaces = set(t['database'] for t in inventaire)

for ns in namespaces:
    # Création du namespace
    response = requests.post(
        f"{REST_CATALOG_URL}/v1/namespaces",
        headers=HEADERS,
        json={
            "namespace": [ns],
            "properties": {
                "location": f"s3://entreprise-lakehouse/{ns}",
                "migrated_from": "hive_metastore",
                "migration_date": "2024-01-15"
            }
        }
    )

    if response.status_code == 200:
```

```
        print(f"Namespace {ns} créé")
    elif response.status_code == 409:
        print(f"Namespace {ns} existe déjà")
    else:
        print(f"Erreur création {ns}: {response.text}")
```

Phase 3 : Enregistrement des tables

```
# Connexion au nouveau REST Catalog
spark_rest = SparkSession.builder \
    .config("spark.sql.catalog.lakehouse", "org.apache.iceberg.spark.SparkCatalog") \
    .config("spark.sql.catalog.lakehouse.type", "rest") \
    .config("spark.sql.catalog.lakehouse.uri", REST_CATALOG_URL) \
    .getOrCreate()

for table_info in inventaire:
    db = table_info['database']
    tbl = table_info['table']
    metadata_loc = table_info['metadata_location']

    try:
        # Enregistrement via REGISTER TABLE (Iceberg 1.4+)
        spark_rest.sql(f"""
            CALL lakehouse.system.register_table(
                table => '{db}.{tbl}',
                metadata_file => '{metadata_loc}'
            )
        """)
        print(f"Table {db}.{tbl} enregistrée")

    except Exception as e:
        print(f"Erreur enregistrement {db}.{tbl}: {e}")
        # Journalisation pour reprise
        log_migration_error(db, tbl, str(e))
```

Phase 4 : Validation

```
# Validation de la migration
def validate_table_migration(db, tbl):
    # Comparaison des métadonnées
    hive_meta = spark_hive.sql(f"""
        SELECT COUNT(*) as cnt, SUM(file_size_in_bytes) as size
        FROM hive_catalog.{db}.{tbl}.files
    """).collect()[0]

    rest_meta = spark_rest.sql(f"""
        SELECT COUNT(*) as cnt, SUM(file_size_in_bytes) as size
        FROM lakehouse.{db}.{tbl}.files
    """).collect()[0]

    if hive_meta.cnt == rest_meta.cnt and hive_meta.size == rest_meta.size:
        print(f"✓ {db}.{tbl} validée")
        return True
    else:
        print(f"✗ {db}.{tbl} divergente: HMS({hive_meta.cnt}, {hive_meta.size}) vs
REST({rest_meta.cnt}, {rest_meta.size})")
```

```
        return False

# Validation de toutes les tables
resultats = [validate_table_migration(t['database'], t['table']) for t in inventaire]
print(f"Migration validée: {sum(resultats)}/{len(resultats)} tables")
```

Migration Glue vers REST Catalog

Pour les organisations souhaitant réduire leur dépendance à AWS, la migration depuis Glue vers un REST Catalog offre plus de flexibilité.

```
import boto3

# Client Glue
glue = boto3.client('glue', region_name='ca-central-1')

# Récupération des tables Iceberg depuis Glue
def get_glue_iceberg_tables(database_name):
    tables = []
    paginator = glue.get_paginator('get_tables')

    for page in paginator.paginate(DatabaseName=database_name):
        for table in page['TableList']:
            # Vérification si c'est une table Iceberg
            params = table.get('Parameters', {})
            if params.get('table_type') == 'ICEBERG':
                tables.append({
                    'database': database_name,
                    'table': table['Name'],
                    'location': table['StorageDescriptor']['Location'],
                    'metadata_location': params.get('metadata_location')
                })

    return tables

# Migration
databases = ['lakehouse_bronze', 'lakehouse_silver', 'lakehouse_gold']
all_tables = []

for db in databases:
    all_tables.extend(get_glue_iceberg_tables(db))

print(f"Tables Glue à migrer: {len(all_tables)}")

# Enregistrement dans REST Catalog (même logique que précédemment)
for table_info in all_tables:
    register_in_rest_catalog(table_info)
```

Études de Cas Canadiennes

Secteur Assurance : Assureur National

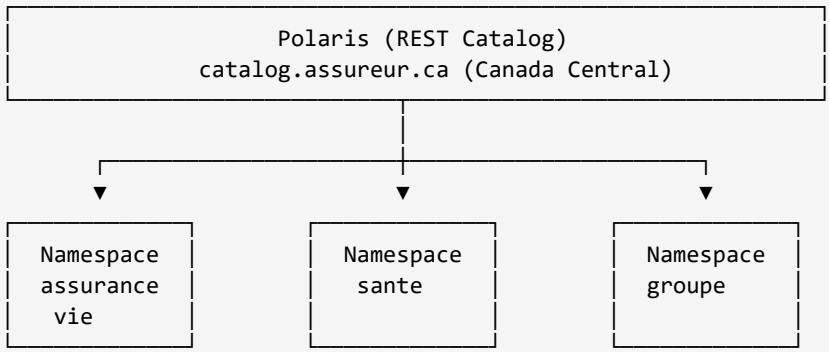
Étude de cas : Grand assureur pancanadien

Secteur : Assurance vie et santé

Défi : Unifier les catalogues de données hérités de 5 acquisitions tout en respectant les exigences de l'AMF (Autorité des marchés financiers) et du BSIF concernant la gouvernance des données. Chaque entité acquise opérait son propre Hive Metastore avec des conventions de nommage incompatibles.

Solution : Déploiement d'Apache Polaris comme REST Catalog central dans la région Azure Canada Central. Migration progressive des 5 Hive Metastores sur 8 mois. Implémentation d'une couche de gouvernance avec Azure Purview pour la découverte et le lignage.

Architecture :



Résultats :

- Consolidation de 5 catalogues en 1 avec 2 400 tables migrées
- Temps de découverte des données réduit de 85%
- Conformité aux exigences de traçabilité de l'AMF
- Économies de 400 000$/an en coûts d'infrastructure HMS

Secteur Minier : Société d'Exploration

Étude de cas : Société d'exploration minière québécoise

Secteur : Ressources naturelles et exploitation minière

Défi : Gérer les données de 12 sites d'exploration répartis dans des régions éloignées du Nord québécois avec connectivité limitée. Les données géologiques et de forage devaient être consolidées pour l'analyse centrale tout en permettant l'autonomie locale.

Solution : Architecture de catalogue distribuée avec Nessie déployé localement sur chaque site (MinIO + Nessie sur serveurs edge) et synchronisation périodique vers le catalogue central hébergé dans le centre

de données de Montréal. Les branches Nessie permettent aux géologues de travailler hors ligne puis de synchroniser.

Particularités :

- Connectivité satellite intermittente (1-4 Mbps)
- Synchronisation par batch quotidien via Starlink
- Résolution de conflits automatisée pour les données de forage

Résultats :

- Analyse consolidée de 15 ans de données d'exploration
- Productivité des géologues de terrain augmentée de 40%
- Résilience totale aux pannes de connectivité

Secteur Public : Ministère Provincial

Étude de cas : Ministère de la Santé provincial

Secteur : Gouvernement provincial

Défi : Moderniser l'infrastructure de données tout en maintenant une stricte conformité aux exigences de la Loi 25 (protection des renseignements personnels). Les données de santé ne pouvaient en aucun cas résider hors du Canada ou transiter par des services américains.

Solution : Déploiement entièrement sur site avec MinIO pour le stockage et REST Catalog (implémentation maison basée sur la spécification Iceberg) hébergé dans les centres de données gouvernementaux certifiés. Intégration avec le système d'identité gouvernemental (CLIQ) pour l'authentification.

Architecture de sécurité :

- Chiffrement AES-256 des données au repos et en transit
- Audit exhaustif de toutes les opérations catalogue
- Ségrégation réseau entre environnements
- Authentification multifacteur obligatoire

Résultats :

- Conformité Loi 25 validée par audit externe
- Aucune donnée transitant hors des infrastructures gouvernementales
- Temps de provisionnement d'accès réduit de 2 semaines à 2 heures

Bonnes Pratiques et Recommandations

Convention de Nommage

Une convention de nommage cohérente simplifie la découverte et la gouvernance des données à l'échelle de l'entreprise.

Structure hiérarchique recommandée :

```
{catalogue}.{couche}.{domaine}_{entité}_{qualificateur}

Exemples:
lakehouse.bronze.ventes_transactions_raw
lakehouse.silver.ventes_transactions_clean
lakehouse.gold.ventes_kpi_quotidien
lakehouse.gold.ventes_cube_regional
```

Règles de nommage :

Élément	Convention	Exemples
Catalogue	snake_case, environnement	lakehouse_prod, lakehouse_dev
Namespace	snake_case, couche/domaine	bronze, silver, gold, ventes, rh
Table	snake_case, entité_qualificateur	transactions_raw, clients_v2
Colonne	snake_case	client_id, date_transaction

Haute Disponibilité

Configuration REST Catalog HA :

```yaml
# Déploiement multi-zone avec réplication
apiVersion: apps/v1
kind: Deployment
metadata:
  name: polaris-catalog
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchLabels:
                  app: polaris-catalog
              topologyKey: topology.kubernetes.io/zone
```

```
      containers:
        - name: polaris
          # ... configuration ...

---
# Base de données PostgreSQL en haute disponibilité
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: polaris-db
spec:
  instances: 3
  primaryUpdateStrategy: unsupervised

  storage:
    size: 100Gi
    storageClass: gp3

  backup:
    barmanObjectStore:
      destinationPath: s3://entreprise-backups/polaris-db
      s3Credentials:
        accessKeyId:
          name: s3-creds
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: s3-creds
          key: SECRET_ACCESS_KEY
      wal:
        compression: gzip
    retentionPolicy: "30d"
```

Surveillance et Alertes

Métriques essentielles :

Métrique	Seuil d'alerte	Description
catalog_request_latency_p99	> 500ms	Latence des requêtes catalogue
catalog_error_rate	> 1%	Taux d'erreurs des opérations
catalog_active_connections	> 80% capacité	Connexions actives
catalog_commit_conflicts	> 10/min	Conflits de commit
catalog_table_count	Croissance > 10%/jour	Prolifération des tables

Configuration Prometheus :

```
# prometheus-rules.yaml
groups:
  - name: catalog-alerts
    rules:
      - alert: CatalogLatencyHigh
        expr: histogram_quantile(0.99, rate(catalog_request_duration_seconds_bucket[5m]))
```

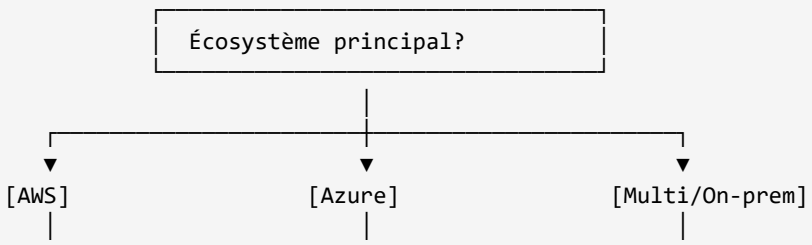
```
> 0.5
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "Latence catalogue élevée"
          description: "P99 latence > 500ms depuis 5 minutes"

      - alert: CatalogErrorRateHigh
        expr: rate(catalog_requests_total{status="error"}[5m]) /
rate(catalog_requests_total[5m]) > 0.01
        for: 5m
        labels:
          severity: critical
        annotations:
          summary: "Taux d'erreurs catalogue critique"

      - alert: CatalogCommitConflicts
        expr: rate(catalog_commit_conflicts_total[5m]) > 10/60
        for: 10m
        labels:
          severity: warning
        annotations:
          summary: "Conflits de commit fréquents"
          description: "Vérifier la charge d'écriture concurrente"
```

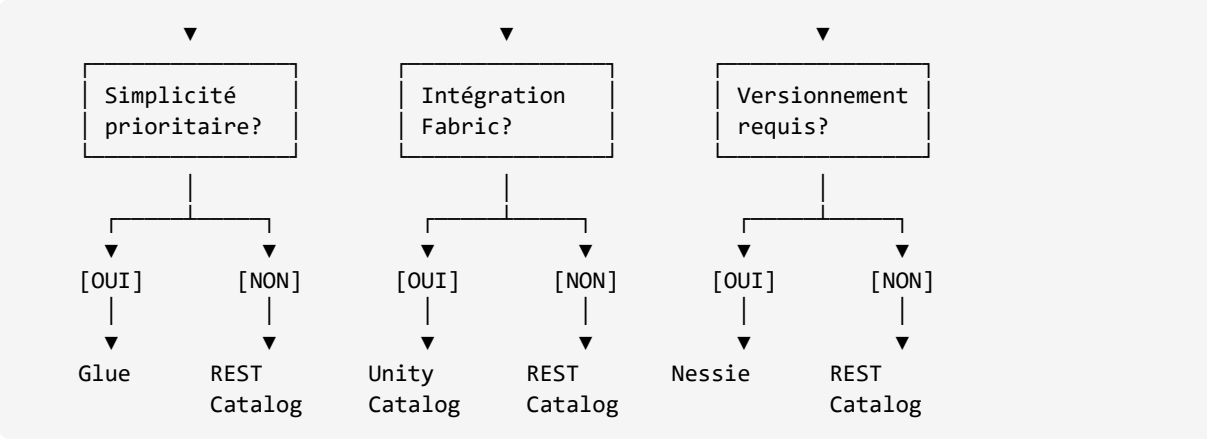
Matrice de Décision

Choix du type de catalogue :

Critère	Hive MS	REST Catalog	Glue	Nessie
Écosystème existant Hadoop	★★★★★	★★★☆☆	★★☆☆☆	★★☆☆☆
Multi-moteur moderne	★★★☆☆	★★★★★	★★★★☆	★★★★☆
Complexité opérationnelle	★★★☆☆	★★☆☆☆	★☆☆☆☆	★★★☆☆
Gouvernance native	★★☆☆☆	★★★★☆	★★★★★	★★★☆☆
Versionnement Git-like	☆☆☆☆☆	☆☆☆☆☆	☆☆☆☆☆	★★★★★
Verrouillage fournisseur	★☆☆☆☆	★☆☆☆☆	★★★★★	★☆☆☆☆
Coût (grande échelle)	★★★☆☆	★★★☆☆	★★☆☆☆	★★★★☆

Arbre de décision :





Conclusion

La couche de catalogue constitue le système nerveux central de votre Data Lakehouse Apache Iceberg. Elle transforme une collection de fichiers Parquet en une plateforme de données gouvernée, découvrable et accessible à l'ensemble de l'organisation. Le choix et l'implémentation de cette couche influencent directement l'expérience utilisateur, la sécurité des données et l'agilité opérationnelle de votre plateforme.

L'écosystème des catalogues Iceberg a considérablement mûri ces dernières années. Du traditionnel Hive Metastore aux solutions modernes comme le REST Catalog et Nessie, les organisations disposent désormais d'options adaptées à chaque contexte : services gérés pour la simplicité opérationnelle, solutions open source pour le contrôle total, catalogues Git-like pour les pratiques DevOps avancées.

Pour les organisations canadiennes, les considérations de résidence des données et de conformité réglementaire ajoutent une dimension critique au choix du catalogue. Les études de cas présentées démontrent que des architectures conformes aux exigences les plus strictes — Loi 25, AMF, BSIF — sont réalisables avec les technologies actuelles, que ce soit via des services infonuagiques dans les régions canadiennes ou des déploiements entièrement sur site.

La migration entre catalogues, longtemps perçue comme risquée, devient une opération de routine grâce à l'architecture découplée d'Iceberg. Les métadonnées détaillées résidant dans le stockage objet, seuls les pointeurs doivent être recréés dans le nouveau catalogue. Cette portabilité libère les organisations du verrouillage technologique et leur permet de faire évoluer leur architecture au rythme de leurs besoins.

Le chapitre suivant explore la couche de fédération et de requête, où nous examinerons comment les moteurs analytiques comme Trino et Dremio consomment les métadonnées du catalogue pour exécuter des requêtes performantes sur votre Lakehouse Iceberg.

Résumé

Rôle du catalogue Iceberg :

- Registre des tables avec pointeurs vers les métadonnées
- Gestion des espaces de noms pour l'organisation logique
- Contrôle de concurrence pour les commits atomiques

- Interface standardisée indépendante de l'implémentation

Principales implémentations :

- **Hive Metastore** : Maturité et compatibilité Hadoop, mais fonctionnalités limitées
- **REST Catalog** : Standard moderne, interopérabilité maximale, déploiement requis
- **AWS Glue** : Service géré, intégration AWS native, verrouillage fournisseur
- **Nessie** : Versionnement Git-like, branches et merges de données

Sécurité et gouvernance :

- Authentification OAuth 2.0/OIDC pour intégration IAM
- Contrôle d'accès RBAC au niveau catalogue, namespace et table
- Masquage des données sensibles via politiques
- Audit exhaustif pour traçabilité et conformité

Migration :

- Réenregistrement des tables sans déplacement de données
- Validation par comparaison des métadonnées
- Migration progressive avec période de coexistence

Recommandations :

- Convention de nommage cohérente : catalogue.couche.domaine_entité
- Haute disponibilité : déploiement multi-zone, base de données répliquée
- Surveillance : latence, taux d'erreurs, conflits de commit
- Choix selon contexte : Glue pour AWS, REST pour multi-moteur, Nessie pour DevOps

---

Ce chapitre établit la gouvernance des métadonnées de votre Lakehouse. Le chapitre suivant, « Requêtes Fédérées et Moteurs de Consommation », détaille comment exploiter efficacement ces métadonnées pour l'analytique à l'échelle de l'entreprise.

Chapitre IV.8 - Conception de la Couche de Fédération

Introduction

La couche de fédération représente l'interface stratégique entre votre Data Lakehouse Apache Iceberg et les consommateurs de données — analystes, scientifiques de données, applications et tableaux de bord. Elle unifie l'accès à des sources hétérogènes, masque la complexité technique sous-jacente et optimise l'exécution des requêtes pour offrir des performances analytiques de classe entreprise. Sans cette couche, votre Lakehouse demeure une infrastructure technique ; avec elle, il devient une plateforme de données accessible et exploitable par l'ensemble de l'organisation.

Dans l'architecture traditionnelle des entrepôts de données, la fédération se limitait souvent à connecter quelques sources via des liens de base de données. L'écosystème moderne du Lakehouse élargit considérablement cette vision : les moteurs de requête fédérée comme Trino, Dremio ou Spark SQL permettent d'interroger simultanément des tables Iceberg, des bases de données relationnelles, des API REST, des fichiers CSV et des services infonuagiques — le tout via une syntaxe SQL unifiée et des optimisations transparentes.

Ce chapitre vous guide dans la conception et l'implémentation d'une couche de fédération performante pour votre Lakehouse Iceberg. Nous examinerons les moteurs de requête disponibles, leurs forces respectives et leurs cas d'usage optimaux. Nous détaillerons les configurations de production, les stratégies d'optimisation des performances et les patterns architecturaux éprouvés. Les études de cas canadiennes illustreront comment des organisations de divers secteurs ont déployé ces technologies pour transformer leurs capacités analytiques.

L'enjeu dépasse la simple exécution de requêtes SQL. La couche de fédération définit l'expérience utilisateur de votre plateforme de données, influence les coûts d'exploitation et détermine la capacité de votre organisation à démocratiser l'accès aux données tout en maintenant une gouvernance rigoureuse.

Fondamentaux de la Fédération de Données

Définition et Objectifs

La fédération de données désigne la capacité d'accéder à des données distribuées sur plusieurs systèmes comme si elles résidaient dans une source unique. Cette abstraction offre plusieurs bénéfices stratégiques :

Accès unifié : Les utilisateurs interrogent les données via une interface SQL standard, indépendamment de leur localisation physique ou de leur format de stockage. Un analyste peut joindre une table Iceberg avec une base PostgreSQL et un fichier CSV sans connaître les détails techniques de chaque source.

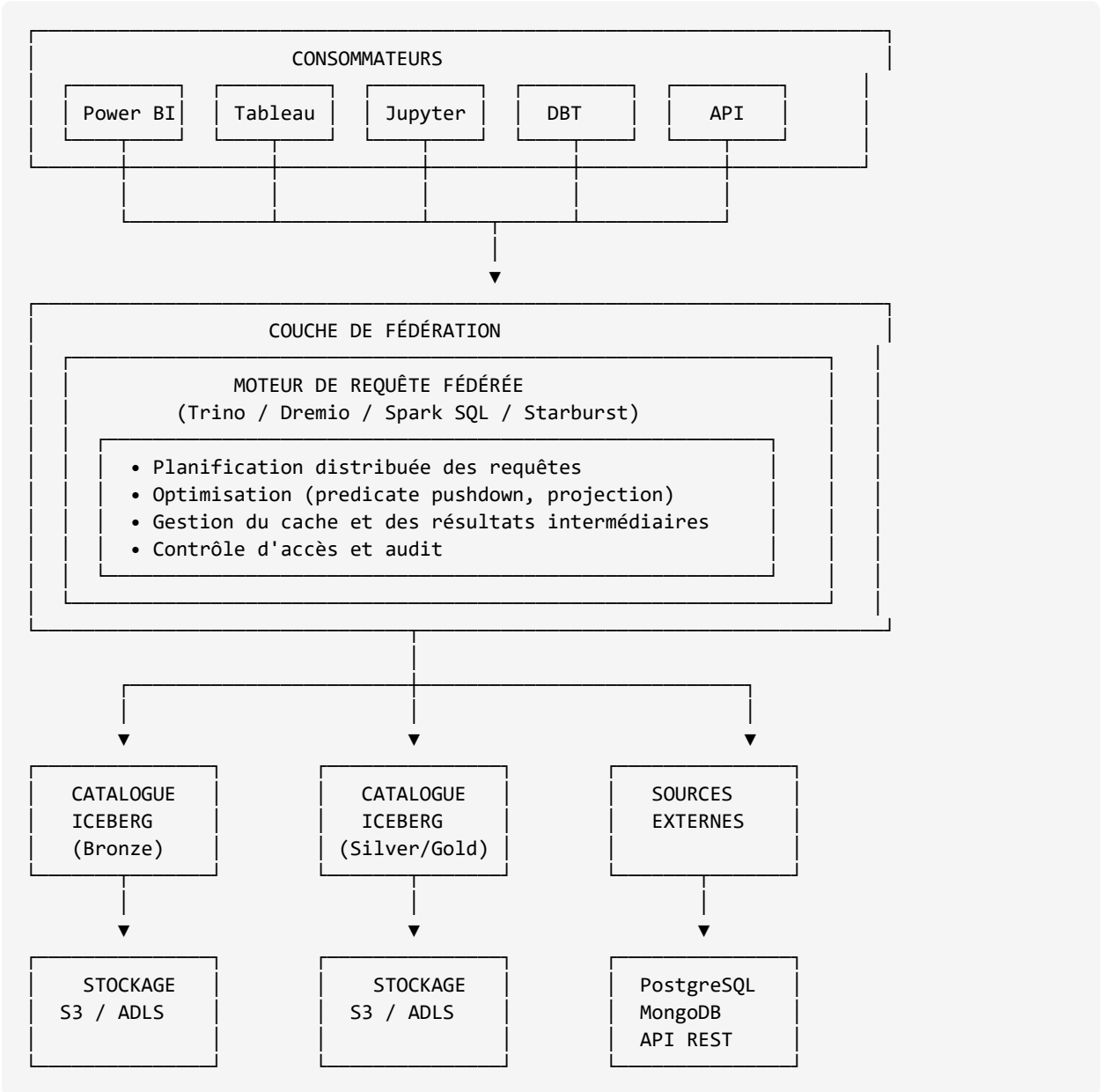
Réduction de la duplication : Plutôt que de copier les données vers un entrepôt central, la fédération permet de les interroger in situ. Cette approche réduit les coûts de stockage, élimine les problèmes de synchronisation et garantit l'accès aux données les plus récentes.

Agilité analytique : Les nouvelles sources de données deviennent accessibles rapidement, sans les délais traditionnels d'intégration ETL. Cette agilité accélère les cycles d'analyse et permet de répondre plus rapidement aux besoins métier.

Gouvernance centralisée : La couche de fédération devient le point de contrôle unique pour l'accès aux données, simplifiant l'application des politiques de sécurité et d'audit.

Architecture de Fédération pour Iceberg

L'architecture de fédération pour un Lakehouse Iceberg s'articule autour de plusieurs composantes inter-dépendantes :



Patterns de Requêtes Fédérées

La fédération de données supporte plusieurs patterns d'accès, chacun avec ses caractéristiques de performance et ses cas d'usage :

Pattern 1 : Requête locale Iceberg La requête cible exclusivement des tables Iceberg du même catalogue. C'est le scénario le plus performant, bénéficiant pleinement des optimisations Iceberg.

```
-- Requête locale optimisée
SELECT
    date_transaction,
    SUM(montant) as total_ventes
FROM lakehouse.gold.transactions
WHERE date_transaction >= DATE '2024-01-01'
GROUP BY date_transaction;
```

Pattern 2 : Jointure inter-catalogue La requête joint des tables de différents catalogues Iceberg, potentiellement sur différents systèmes de stockage.

```
-- Jointure entre catalogues
SELECT
    t.transaction_id,
    c.nom_client,
    t.montant
FROM lakehouse_ventes.gold.transactions t
JOIN lakehouse_crm.silver.clients c
    ON t.client_id = c.client_id
WHERE t.date_transaction >= DATE '2024-01-01';
```

Pattern 3 : Fédération hybride La requête combine des données Iceberg avec des sources externes (SGBD, fichiers, API).

```
-- Fédération avec source externe
SELECT
    i.transaction_id,
    i.montant,
    p.nom_produit,
    inv.quantite_stock
FROM lakehouse.gold.transactions i
JOIN postgresql.erp.produits p
    ON i.produit_id = p.produit_id
JOIN mongodb.inventaire.stock inv
    ON p.sku = inv.sku
WHERE i.date_transaction = CURRENT_DATE;
```

Pattern 4 : Virtualisation de données Création de vues virtuelles combinant plusieurs sources, exposées comme une source unique aux consommateurs.

```
-- Vue virtualisée
CREATE VIEW unified.analytics.vue_360_client AS
SELECT
    c.client_id,
    c.nom,
    c.segment,
    COALESCE(t.total_achats, 0) as total_achats,
```

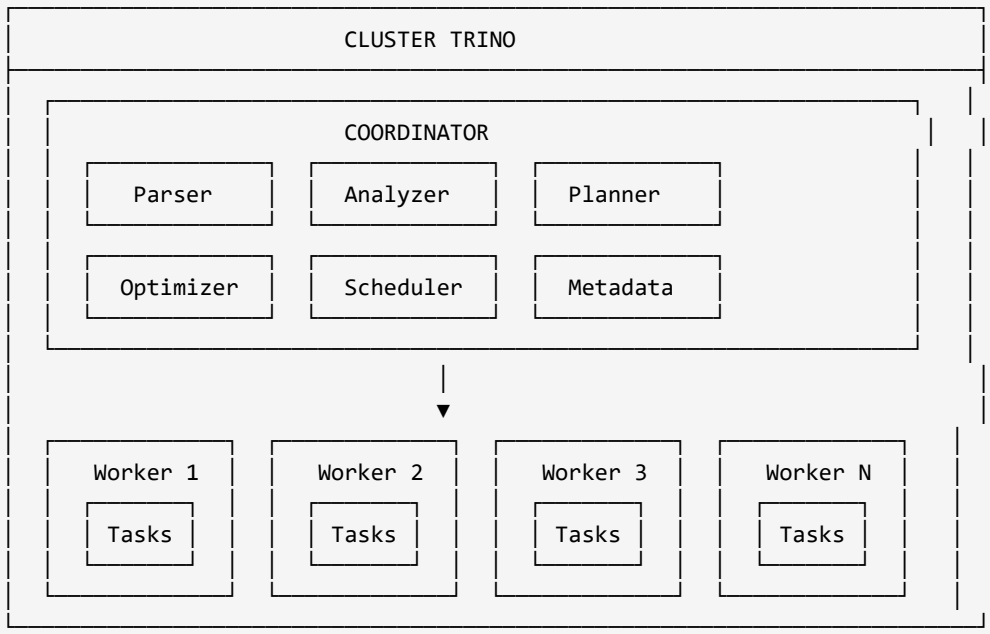
```
    COALESCE(s.score_satisfaction, 0) as satisfaction,
    COALESCE(w.derniere_visite, DATE '1900-01-01') as derniere_visite_web
FROM lakehouse.gold.clients c
LEFT JOIN lakehouse.gold.transactions_agregees t
    ON c.client_id = t.client_id
LEFT JOIN salesforce.survey.scores s
    ON c.courriel = s.email
LEFT JOIN snowplow.analytics.sessions w
    ON c.client_id = w.user_id;
```

Moteurs de Requête Fédérée

Trino : Le Standard Open Source

Trino (anciennement Presto SQL) s'est établi comme le moteur de requête fédérée de référence dans l'écosystème open source. Conçu pour les requêtes analytiques interactives à grande échelle, il offre une architecture distribuée optimisée pour la latence faible et le débit élevé.

Architecture Trino :



Forces de Trino :

- Architecture MPP (Massively Parallel Processing) pour performances élevées
- Écosystème de connecteurs riche (40+ sources supportées)
- Communauté active et développement continu
- Coût nul de licence (open source Apache 2.0)
- Intégration native avec Iceberg de haute qualité

Limitations :

- Pas de persistance des résultats intermédiaires (tout en mémoire)
- Configuration et tuning requièrent une expertise
- Moins adapté aux requêtes très longues (heures)

Configuration Trino pour Iceberg :

```
# etc/catalog/lakehouse.properties
connector.name=iceberg

# Configuration du catalogue
iceberg.catalog.type=rest
iceberg.rest-catalog.uri=https://catalog.lakehouse.entreprise.ca
iceberg.rest-catalog.warehouse=s3://entreprise-lakehouse/warehouse

# Authentification
iceberg.rest-catalog.security=OAUTH2
iceberg.rest-catalog.oauth2.credential=file:///etc/trino/secrets/oauth-credential

# Configuration S3
iceberg.file-system-type=S3
s3.region=ca-central-1
s3.endpoint=s3.ca-central-1.amazonaws.com
s3.path-style-access=false

# Optimisations de lecture
iceberg.split-manager-threads=32
iceberg.max-partitions-per-writer=1000
iceberg.target-max-file-size=512MB

# Cache de métadonnées
iceberg.metadata.cache-ttl=5m
iceberg.metadata.cache-size=1000

# Optimisations de performance
iceberg.parquet.use-column-index=true
iceberg.parquet.use-bloom-filter=true
```

Déploiement Kubernetes :

```
# trino-deployment.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: trino-config
  namespace: lakehouse
data:
  config.properties: |
    coordinator=true
    node-scheduler.include-coordinator=false
    http-server.http.port=8080
    discovery.uri=http://trino-coordinator:8080
    query.max-memory=50GB
    query.max-memory-per-node=8GB
    query.max-total-memory-per-node=10GB
    memory.heap-headroom-per-node=2GB

  jvm.config: |
    -server
```

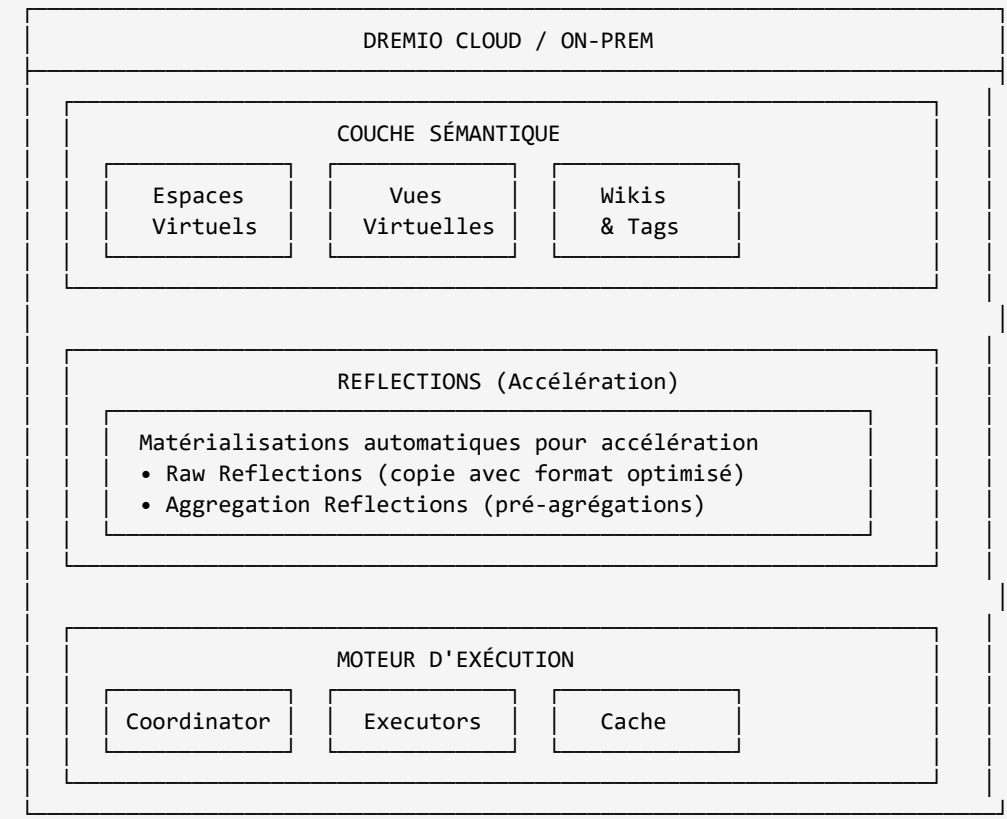
```
    -Xmx16G
    -XX:+UseG1GC
    -XX:G1HeapRegionSize=32M
    -XX:+ExplicitGCInvokesConcurrent
    -XX:+HeapDumpOnOutOfMemoryError
    -XX:HeapDumpPath=/var/trino/data
    -Djdk.attach.allowAttachSelf=true
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: trino-coordinator
  namespace: lakehouse
spec:
  replicas: 1
  selector:
    matchLabels:
      app: trino
      role: coordinator
  template:
    metadata:
      labels:
        app: trino
        role: coordinator
    spec:
      containers:
        - name: trino
          image: trinodb/trino:435
          ports:
            - containerPort: 8080
          resources:
            requests:
              memory: "20Gi"
              cpu: "4"
            limits:
              memory: "24Gi"
              cpu: "8"
          volumeMounts:
            - name: config
              mountPath: /etc/trino
            - name: catalog
              mountPath: /etc/trino/catalog
      volumes:
        - name: config
          configMap:
            name: trino-config
        - name: catalog
          configMap:
            name: trino-catalogs
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: trino-worker
  namespace: lakehouse
spec:
  replicas: 10
  selector:
    matchLabels:
```

```
      app: trino
      role: worker
  template:
    metadata:
      labels:
        app: trino
        role: worker
    spec:
      containers:
        - name: trino
          image: trinodb/trino:435
          resources:
            requests:
              memory: "32Gi"
              cpu: "8"
            limits:
              memory: "40Gi"
              cpu: "16"
```

Dremio : La Plateforme Lakehouse Unifiée

Dremio se positionne comme une plateforme Lakehouse complète, combinant moteur de requête, couche sémantique et accélération des requêtes. Son intégration native avec Iceberg et ses fonctionnalités d'entreprise en font une option attractive pour les organisations recherchant une solution clé en main.

Architecture Dremio :



Forces de Dremio :

- Reflections : accélération automatique des requêtes via matérialisation
- Couche sémantique intégrée pour gouvernance et collaboration
- Interface utilisateur intuitive pour exploration des données
- Support natif d'Iceberg avec optimisations spécifiques
- Mode SaaS (Dremio Cloud) pour déploiement simplifié

Limitations :

- Coût de licence significatif pour les fonctionnalités entreprise
- Complexité de dimensionnement des Reflections
- Moins de connecteurs que Trino pour sources exotiques

Configuration des sources Iceberg dans Dremio :

```
-- Ajout d'une source Iceberg Nessie
ALTER SOURCE lakehouse_bronze SET (
    TYPE = 'NESSIE',
    NESSIE_ENDPOINT = 'http://nessie:19120/api/v2',
    NESSIE_AUTH_TYPE = 'BEARER',
    NESSIE_AUTH_TOKEN = '${NESSIE_TOKEN}',
    NESSIE_DEFAULT_BRANCH = 'main',
    AWS_ROOT_PATH = 's3://entreprise-lakehouse/bronze',
    AWS_REGION = 'ca-central-1',
    ENABLE_ASYNC_ACCESS = true
);

-- Ajout d'une source Iceberg AWS Glue
ALTER SOURCE lakehouse_gold SET (
    TYPE = 'AWSGLUE',
    GLUE_CATALOG_ID = '123456789012',
    GLUE_AWS_REGION = 'ca-central-1',
    GLUE_DATABASE_FILTER = 'lakehouse_gold.*',
    AWS_ROOT_PATH = 's3://entreprise-lakehouse/gold'
);
```

Configuration des Reflections :

```
-- Création d'une Raw Reflection pour accélération de lecture
ALTER TABLE lakehouse_gold.transactions
CREATE RAW REFLECTION reflection_transactions_raw
USING DISPLAY (
    transaction_id,
    client_id,
    produit_id,
    montant,
    date_transaction,
    region
)
PARTITION BY (TRUNCATE(date_transaction, MONTH), region)
LOCALSORT BY (client_id);

-- Création d'une Aggregation Reflection pour accélération d'agrégats
ALTER TABLE lakehouse_gold.transactions
CREATE AGGREGATION REFLECTION reflection_transactions_agg
USING DIMENSIONS (
    date_transaction,
    region,
```

```
    categorie_produit
)
MEASURES (
    montant (SUM, COUNT, MIN, MAX),
    quantite (SUM, COUNT)
)
PARTITION BY (TRUNCATE(date_transaction, MONTH));
```

Apache Spark SQL

Apache Spark SQL offre des capacités de requête fédérée intégrées au framework Spark, particulièrement adaptées aux pipelines de transformation et aux requêtes longues.

Forces de Spark SQL :

- Intégration native avec les pipelines Spark existants
- Excellent pour les requêtes longues et les transformations complexes
- Support DataFrame/Dataset pour développement programmatique
- Écosystème MLlib pour analytics avancé

Limitations :

- Latence de démarrage (cold start) élevée
- Moins optimisé pour les requêtes ad-hoc interactives
- Overhead de gestion de cluster

Configuration Spark SQL pour fédération :

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("FederatedAnalytics") \
    .config("spark.sql.extensions",
            "org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions") \
    \
    # Catalogue Iceberg principal
    .config("spark.sql.catalog.lakehouse",
            "org.apache.iceberg.spark.SparkCatalog") \
    .config("spark.sql.catalog.lakehouse.type", "rest") \
    .config("spark.sql.catalog.lakehouse.uri",
            "https://catalog.lakehouse.entreprise.ca") \
    \
    # Source PostgreSQL via JDBC
    .config("spark.sql.catalog.postgres",
            "org.apache.spark.sql.execution.datasources.v2.jdbc.JDBCTableCatalog") \
    .config("spark.sql.catalog.postgres.url",
            "jdbc:postgresql://postgres-erp:5432/production") \
    .config("spark.sql.catalog.postgres.driver",
            "org.postgresql.Driver") \
    \
    # Source MongoDB
    .config("spark.mongodb.read.connection.uri",
            "mongodb://mongo-cluster:27017") \
    .config("spark.mongodb.read.database", "analytics") \
    \
    .getOrCreate()
```

```
# Requête fédérée
df_federated = spark.sql("""
    SELECT
        i.transaction_id,
        i.montant,
        p.nom_produit,
        m.score_satisfaction
    FROM lakehouse.gold.transactions i
    JOIN postgres.public.produits p
        ON i.produit_id = p.id
    LEFT JOIN mongo_analytics.feedback m
        ON i.transaction_id = m.transaction_id
    WHERE i.date_transaction >= '2024-01-01'
""")
```

Starburst Galaxy et Enterprise

Starburst, l'entreprise derrière le fork commercial de Trino, offre des solutions entreprise avec fonctionnalités avancées de sécurité, gouvernance et support.

Starburst Galaxy (SaaS) :

- Déploiement géré dans le nuage
- Intégration native avec les services infonuagiques
- Découverte automatique des données
- Interface de requête intégrée

Starburst Enterprise (On-premise/Hybride) :

- Contrôle d'accès basé sur les rôles avancé
- Audit et lignage des données
- Support et SLA entreprise
- Intégration LDAP/AD native

Configuration Starburst avec BIAC (Built-in Access Control) :

```
# etc/catalog/lakehouse.properties avec contrôle d'accès
connector.name=iceberg
iceberg.catalog.type=rest

# Contrôle d'accès au niveau colonne
iceberg.security=system

# Masquage des données
column.mask.functions.enabled=true
```

```
-- Politique de masquage Starburst
CREATE FUNCTION mask_courriel(val VARCHAR)
RETURNS VARCHAR
RETURN CASE
    WHEN is_member_of('data_admin') THEN val
    ELSE regexp_replace(val, '(.{2}).*@', '$1***@')
END;
```

```
-- Application du masquage
ALTER TABLE lakehouse.silver.clients
ALTER COLUMN courriel SET MASK mask_courriel;
```

Comparaison des Moteurs

Critère	Trino	Dremio	Spark SQL	Star-burst
Latence requêtes	Faible	Très faible (avec Reflections)	Moyenne-Élevée	Faible
Requêtes longues	Limité	Bon	Excellent	Limité
Coût	Gratuit			

| *Gratuit* |

$$ | | | | **Facilité déploiement** | Moyenne | Facile (Cloud) | Complexe | Facile | | **Accélération** | Non native | Reflections | Non native | Non native | | **Couche sémantique** | Non | Oui | Non | Limitée | | **Connecteurs** | 40+ | 30+ | 20+ | 40+ | | **Support Iceberg** | Excellent | Excellent | Excellent | Excellent |

Configuration Avancée des Connecteurs

Connecteur Iceberg Optimisé

La configuration du connecteur Iceberg influence directement les performances de vos requêtes. Une configuration soignée peut améliorer les temps de réponse de 2× à 10×.

Optimisations de lecture :

```
# etc/catalog/lakehouse.properties (Trino)

# Parallélisme de lecture
iceberg.split-manager-threads=32
iceberg.max-splits-per-task=256

# Utilisation des index Parquet
iceberg.parquet.use-column-index=true
iceberg.parquet.use-bloom-filter=true
iceberg.parquet.max-read-block-size=16MB

# Cache de métadonnées agressif
iceberg.metadata.cache-ttl=10m
iceberg.metadata.cache-size=2000
iceberg.manifest-cache-size=5000

# Pushdown de prédicats
iceberg.projection-pushdown-enabled=true
iceberg.statistics-based-optimization-enabled=true
```

```
# Lecture optimisée des petits fichiers
iceberg.merge-small-files-on-read=true
iceberg.small-file-threshold=64MB
```

Configuration pour requêtes de jointure :

```
# Optimisation des jointures distribuées
join-distribution-type=AUTOMATIC
join-reordering-strategy=AUTOMATIC
optimizer.join-max-broadcast-table-size=100MB

# Hash partitioning pour grandes jointures
hash-partition-count=64
```

Connecteurs Sources Externes

PostgreSQL avec pushdown optimisé :

```
# etc/catalog/postgres_erp.properties
connector.name=postgresql
connection-url=jdbc:postgresql://postgres-erp.internal:5432/production
connection-user=${POSTGRES_USER}
connection-password=${POSTGRES_PASSWORD}

# Pool de connexions
connection-pool.max-size=30
connection-pool.min-size=5

# Pushdown agressif
postgresql.array-mapping=AS_ARRAY
postgresql.experimental.enable-string-pushdown-with-collate=true
postgresql.include-system-tables=false

# Statistiques pour optimisation
jdbc.statistics-enabled=true
jdbc.aggregation-pushdown.enabled=true
jdbc.join-pushdown.enabled=true
jdbc.topn-pushdown.enabled=true
```

MongoDB pour données semi-structurées :

```
# etc/catalog/mongo_analytics.properties
connector.name=mongodb
mongodb.connection-url=mongodb://mongo-cluster.internal:27017
mongodb.credentials=${MONGO_CREDENTIALS}

mongodb.schema-collection=_schema
mongodb.case-insensitive-name-matching=true

# Projection et filtre pushdown
mongodb.projection-pushdown-enabled=true
mongodb.filter-pushdown-enabled=true
```

Elasticsearch pour recherche :

```
# etc/catalog/elastic_logs.properties
connector.name=elasticsearch
elasticsearch.host=elasticsearch.internal
elasticsearch.port=9200
elasticsearch.default-schema-name=logs

elasticsearch.request-timeout=30s
elasticsearch.scroll-size=1000
elasticsearch.scroll-timeout=1m

# Pushdown de recherche
elasticsearch.pushdown-enabled=true
```

Configuration Multi-Catalogue

Pour les architectures fédérant plusieurs catalogues Iceberg, la configuration cohérente assure une expérience utilisateur uniforme.

```
# Catalogue Bronze (données brutes)
# etc/catalog/bronze.properties
connector.name=iceberg
iceberg.catalog.type=rest
iceberg.rest-catalog.uri=https://catalog.lakehouse.entreprise.ca
iceberg.rest-catalog.prefix=bronze
iceberg.rest-catalog.warehouse=s3://entreprise-lakehouse/bronze

# Catalogue Silver (données nettoyées)
# etc/catalog/silver.properties
connector.name=iceberg
iceberg.catalog.type=rest
iceberg.rest-catalog.uri=https://catalog.lakehouse.entreprise.ca
iceberg.rest-catalog.prefix=silver
iceberg.rest-catalog.warehouse=s3://entreprise-lakehouse/silver

# Catalogue Gold (données agrégées)
# etc/catalog/gold.properties
connector.name=iceberg
iceberg.catalog.type=rest
iceberg.rest-catalog.uri=https://catalog.lakehouse.entreprise.ca
iceberg.rest-catalog.prefix=gold
iceberg.rest-catalog.warehouse=s3://entreprise-lakehouse/gold
```

Optimisation des Performances

Predicate Pushdown et Projection Pushdown

L'optimisation la plus critique pour les performances de requêtes fédérées est le pushdown — la capacité de pousser les filtres et les projections vers les sources de données plutôt que de les appliquer après lecture.

Predicate Pushdown :

```
-- Requête avec filtre
SELECT * FROM lakehouse.gold.transactions
WHERE date_transaction >= DATE '2024-01-01'
  AND region = 'QC'
  AND montant > 1000;

-- Sans pushdown: lecture de TOUTES les données, filtrage en mémoire
-- Avec pushdown: Iceberg utilise les statistiques de partition et de fichier
--                pour ne lire que les fichiers pertinents
```

Vérification du pushdown (Trino) :

```
EXPLAIN (TYPE DISTRIBUTED)
SELECT * FROM lakehouse.gold.transactions
WHERE date_transaction >= DATE '2024-01-01'
  AND region = 'QC';

-- Sortie attendue montrant le pushdown:
-- TableScan[table = lakehouse.gold.transactions,
--           constraint = (date_transaction >= 2024-01-01, region = 'QC')]
--   Layout: [transaction_id, client_id, montant, ...]
--   Estimates: {rows: 125000, cpu: ?, memory: ?, network: ?}
```

Projection Pushdown :

```
-- Sélection de colonnes spécifiques
SELECT transaction_id, montant, date_transaction
FROM lakehouse.gold.transactions
WHERE date_transaction >= DATE '2024-01-01';

-- Iceberg lit uniquement les colonnes demandées du fichier Parquet
-- Réduction potentielle de 80-95% du volume de données lues
```

> **Performance**
>
> Sur une table de 10 To avec 50 colonnes, une requête sélectionnant 5 colonnes avec projection pushdown lit environ 1 To au lieu de 10 To. Combiné avec le predicate pushdown utilisant le partitionnement par date, la lecture peut descendre à 50 Go pour une journée de données. L'impact sur les coûts de transfert S3 et les temps de réponse est considérable.

Stratégies de Jointure

Le choix de la stratégie de jointure influence dramatiquement les performances des requêtes fédérées.

Broadcast Join : La petite table est diffusée à tous les workers.

```
-- Forcer un broadcast join (table de référence)
SELECT /*+ BROADCAST(r) */
    t.transaction_id,
    r.nom_region
FROM lakehouse.gold.transactions t
JOIN lakehouse.gold.regions r ON t.region_id = r.region_id;
```

Distributed Hash Join : Les deux tables sont partitionnées par la clé de jointure.

```
-- Pour grandes tables, le hash join est automatiquement choisi
SELECT
    t.transaction_id,
    c.nom_client
FROM lakehouse.gold.transactions t
JOIN lakehouse.gold.clients c ON t.client_id = c.client_id
WHERE t.date_transaction >= DATE '2024-01-01';
```

Sort Merge Join : Les tables sont triées puis fusionnées.

```
-- Efficace si les données sont déjà triées
-- Configuration Trino
SET SESSION join_distribution_type = 'PARTITIONED';
SET SESSION colocated_join = true;
```

Matrice de décision jointure :

Taille Table A	Taille Table B	Stratégie Recommandée
Petite (< 100 Mo)	Grande	Broadcast A
Grande	Petite (< 100 Mo)	Broadcast B
Grande	Grande	Distributed Hash
Grande (triée)	Grande (triée)	Sort Merge

Gestion du Cache

Le cache améliore significativement les performances des requêtes répétitives et des dashboards.

Cache de métadonnées Trino :

```
# Configuration du cache de métadonnées
iceberg.metadata.cache-ttl=10m
iceberg.metadata.cache-size=2000
iceberg.manifest-cache-size=5000

# Cache de statistiques
iceberg.statistics-enabled=true
iceberg.table-statistics-cache-ttl=1h
```

Cache de données Dremio (Reflections) :

```
-- Configuration du rafraîchissement des Reflections
ALTER TABLE lakehouse.gold.transactions
MODIFY REFLECTION reflection_transactions_raw
REFRESH EVERY 1 HOUR;

-- Rafraîchissement incrémental pour tables Iceberg
ALTER TABLE lakehouse.gold.transactions
```

```
MODIFY REFLECTION reflection_transactions_raw
REFRESH INCREMENTAL;
```

Cache externe avec Alluxio :

```
# Configuration Alluxio comme cache pour Trino
# alluxio-site.properties
alluxio.master.mount.table.root.ufs=s3://entreprise-lakehouse/warehouse
alluxio.user.file.metadata.sync.interval=5m
alluxio.user.file.passive.cache.enabled=true
alluxio.user.file.cache.partially.read.block=true

# Intégration Trino
# etc/catalog/lakehouse.properties
hive.cache.enabled=true
hive.cache.location=/alluxio
hive.cache.data-transfer-size-threshold=16MB
```

Dimensionnement et Scaling

Formule de dimensionnement Trino :

```
Mémoire totale workers = Volume données actives × Facteur concurrence × 0.3
Nombre workers = Mémoire totale / Mémoire par worker

Exemple:
- Volume données actives: 5 To
- Requêtes concurrentes: 50
- Mémoire par worker: 64 Go

Mémoire totale = 5000 Go × 50 × 0.3 = 75 000 Go
Nombre workers = 75 000 / 64 ≈ 1 170 workers (pic théorique)

En pratique avec pushdown efficace:
- Données réellement lues: 5% = 250 Go
- Mémoire nécessaire: 250 × 50 × 0.3 = 3 750 Go
- Nombre workers: 3 750 / 64 ≈ 60 workers
```

Auto-scaling Kubernetes :

```
# trino-hpa.yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: trino-worker-hpa
  namespace: lakehouse
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: trino-worker
  minReplicas: 5
  maxReplicas: 50
  metrics:
```

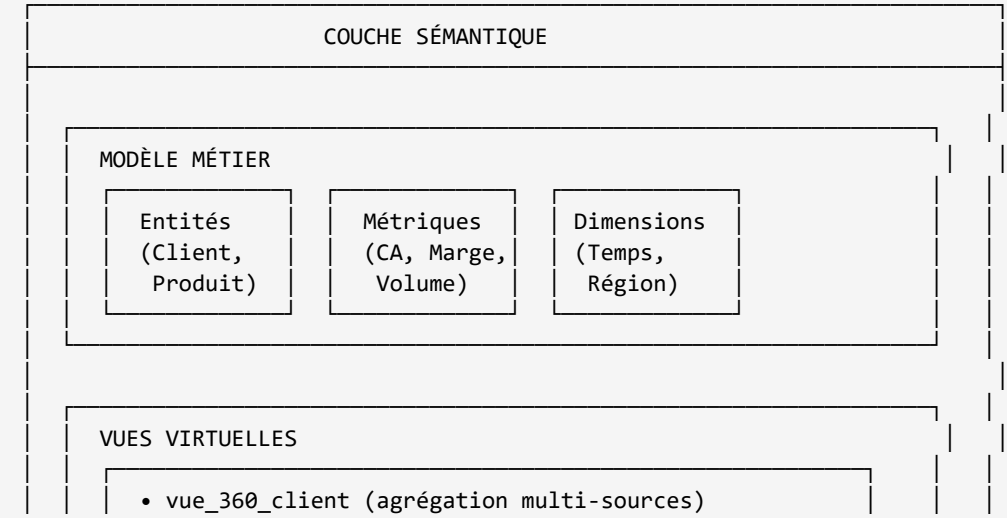
```
    - type: Resource
      resource:
        name: memory
        target:
          type: Utilization
          averageUtilization: 70
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 75
  behavior:
    scaleUp:
      stabilizationWindowSeconds: 60
      policies:
        - type: Pods
          value: 4
          periodSeconds: 60
    scaleDown:
      stabilizationWindowSeconds: 300
      policies:
        - type: Pods
          value: 2
          periodSeconds: 120
```

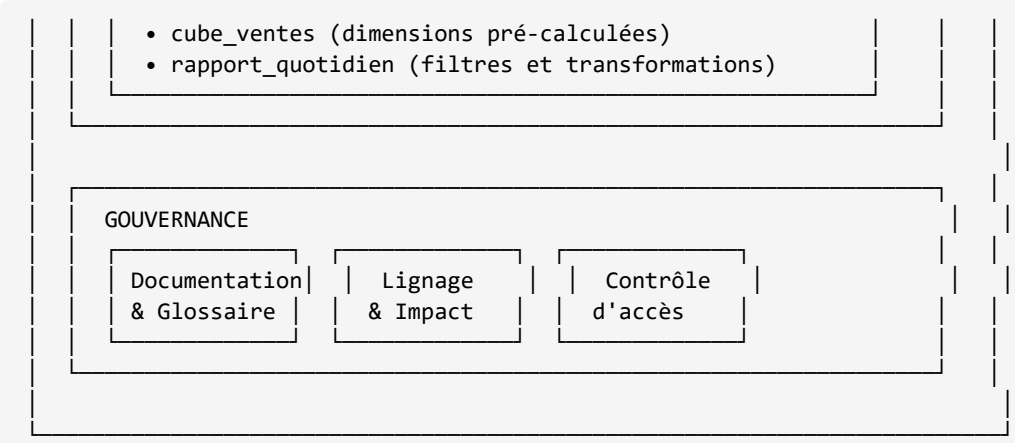
Couche Sémantique et Virtualisation

Concepts de la Couche Sémantique

La couche sémantique ajoute une abstraction métier au-dessus des tables techniques, facilitant la compréhension et l'utilisation des données par les utilisateurs non techniques.

Composantes de la couche sémantique :





Implémentation avec Dremio

Dremio offre une couche sémantique native intégrée à son moteur de requête.

Création d'un espace virtuel :

```
-- Espace pour le domaine Ventes
CREATE SPACE "Analytique Ventes";

-- Dossier pour les vues client
CREATE FOLDER "Analytique Ventes"."Clients";

-- Vue 360° client
CREATE VDS "Analytique Ventes"."Clients"."Vue 360" AS
SELECT
    c.client_id,
    c.nom,
    c.prenom,
    c.courriel,
    c.segment,
    c.date_inscription,

    -- Métriques transactionnelles
    COALESCE(t.nombre_transactions, 0) as nombre_achats,
    COALESCE(t.montant_total, 0) as valeur_totale,
    COALESCE(t.montant_moyen, 0) as panier_moyen,
    t.derniere_transaction,

    -- Score de fidélité calculé
    CASE
        WHEN t.montant_total > 10000 AND t.nombre_transactions > 50 THEN 'Platine'
        WHEN t.montant_total > 5000 AND t.nombre_transactions > 20 THEN 'Or'
        WHEN t.montant_total > 1000 AND t.nombre_transactions > 5 THEN 'Argent'
        ELSE 'Bronze'
    END as niveau_fidelite,

    -- Données comportementales
    w.sessions_30j,
    w.pages_vues_30j,

    -- Satisfaction
    s.score_nps
```

```
FROM lakehouse_gold.clients c

LEFT JOIN (
    SELECT
        client_id,
        COUNT(*) as nombre_transactions,
        SUM(montant) as montant_total,
        AVG(montant) as montant_moyen,
        MAX(date_transaction) as derniere_transaction
    FROM lakehouse_gold.transactions
    WHERE date_transaction >= CURRENT_DATE - INTERVAL '365' DAY
    GROUP BY client_id
) t ON c.client_id = t.client_id

LEFT JOIN lakehouse_silver.web_analytics w
    ON c.client_id = w.user_id

LEFT JOIN salesforce.nps.scores s
    ON c.courriel = s.email;

-- Documentation de la vue
COMMENT ON VDS "Analytique Ventes"."Clients"."Vue 360" IS
'Vue consolidée 360° des clients combinant données transactionnelles,
comportementales et satisfaction. Mise à jour en temps réel.';
```

Configuration des Reflections pour accélération :

```
-- Reflection pour accélération de la vue 360
ALTER VDS "Analytique Ventes"."Clients"."Vue 360"
CREATE RAW REFLECTION reflection_vue360
USING DISPLAY (
    client_id, nom, segment, niveau_fidelite,
    nombre_achats, valeur_totale, panier_moyen,
    score_nps
)
PARTITION BY (segment)
LOCALSORT BY (valeur_totale DESC);

-- Refresh incrémental quotidien
ALTER VDS "Analytique Ventes"."Clients"."Vue 360"
MODIFY REFLECTION reflection_vue360
REFRESH EVERY 24 HOURS;
```

Implémentation avec DBT et Trino

Pour les organisations préférant une approche code-first, DBT combiné à Trino offre une couche sémantique déclarative.

Structure projet DBT :

```
dbt_lakehouse/
├── dbt_project.yml
├── models/
│   ├── staging/
```

```
│   │   ├── stg_transactions.sql
│   │   └── stg_clients.sql
│   ├── intermediate/
│   │   ├── int_transactions_enrichies.sql
│   │   └── int_clients_metriques.sql
│   └── marts/
│       ├── dim_clients.sql
│       ├── dim_produits.sql
│       ├── fct_ventes.sql
│       └── agg_ventes_quotidiennes.sql
├── analyses/
├── macros/
│   └── masquage_pii.sql
└── seeds/
```

Configuration DBT pour Trino-Iceberg :

```yaml
# dbt_project.yml
name: 'lakehouse_analytics'
version: '1.0.0'

profile: 'trino_lakehouse'

model-paths: ["models"]
analysis-paths: ["analyses"]
test-paths: ["tests"]
seed-paths: ["seeds"]
macro-paths: ["macros"]

target-path: "target"
clean-targets:
  - "target"
  - "dbt_packages"

vars:
  date_debut_analyse: '2023-01-01'

models:
  lakehouse_analytics:
    staging:
      +materialized: view
      +schema: staging
    intermediate:
      +materialized: ephemeral
    marts:
      +materialized: table
      +schema: gold
      +file_format: iceberg
      +table_properties:
        write.format.default: 'parquet'
        write.parquet.compression-codec: 'zstd'
```

```yaml
# profiles.yml
trino_lakehouse:
  target: prod
  outputs:
```

```
    prod:
      type: trino
      method: oauth
      host: trino.lakehouse.entreprise.ca
      port: 443
      user: dbt_service
      catalog: lakehouse
      schema: analytics
      threads: 8
      http_scheme: https
      session_properties:
        query_max_execution_time: 2h
```

Modèle DBT pour dimension client :

```
-- models/marts/dim_clients.sql
{{
    config(
        materialized='incremental',
        unique_key='client_id',
        incremental_strategy='merge',
        file_format='iceberg',
        partition_by=['segment'],
        table_properties={
            'write.target-file-size-bytes': '536870912'
        }
    )
}}

WITH clients_base AS (
    SELECT * FROM {{ ref('stg_clients') }}
),

metriques_transactions AS (
    SELECT * FROM {{ ref('int_clients_metriques') }}
),

final AS (
    SELECT
        c.client_id,
        c.nom,
        c.prenom,
        {{ masquer_courriel('c.courriel') }} as courriel,
        c.segment,
        c.province,
        c.date_inscription,

        m.nombre_transactions,
        m.montant_total,
        m.panier_moyen,
        m.derniere_transaction,

        CASE
            WHEN m.montant_total > 10000 THEN 'Platine'
            WHEN m.montant_total > 5000 THEN 'Or'
            WHEN m.montant_total > 1000 THEN 'Argent'
            ELSE 'Bronze'
        END as niveau_fidelite,
```

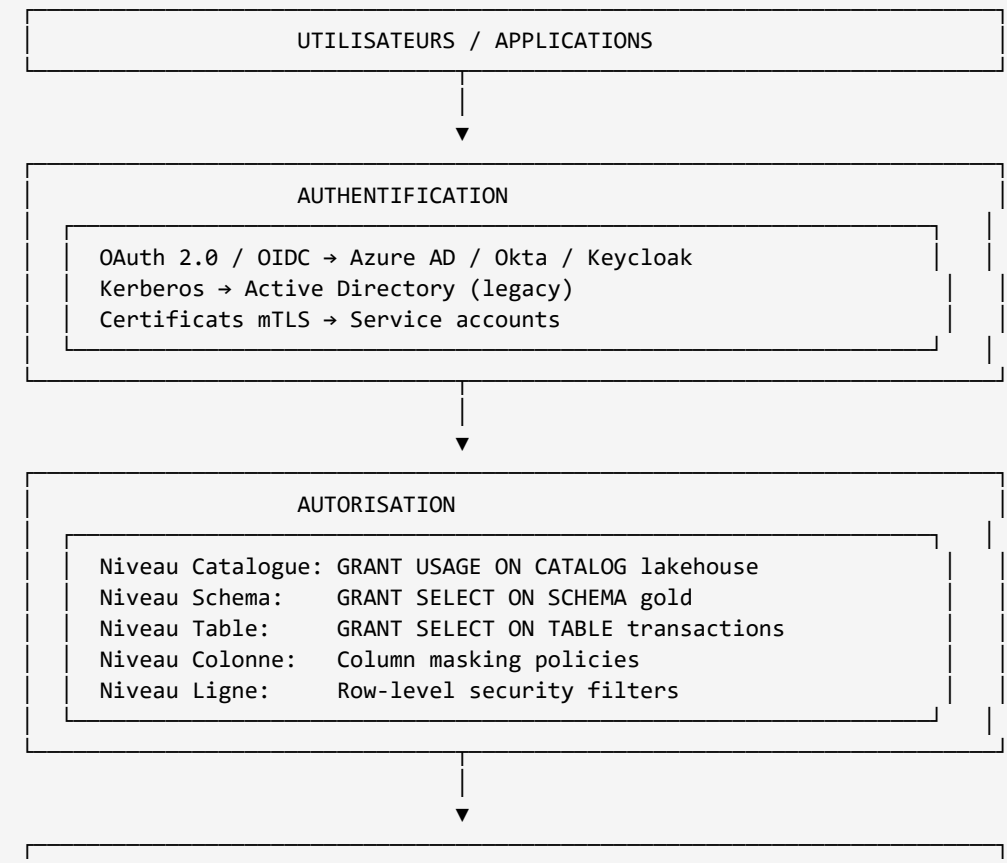
```
        CURRENT_TIMESTAMP as _dbt_updated_at

    FROM clients_base c
    LEFT JOIN metriques_transactions m
        ON c.client_id = m.client_id

    {% if is_incremental() %}
    WHERE c._updated_at > (SELECT MAX(_dbt_updated_at) FROM {{ this }})
       OR m._updated_at > (SELECT MAX(_dbt_updated_at) FROM {{ this }})
    {% endif %}
)

SELECT * FROM final
```

Sécurité et Gouvernance

Contrôle d'Accès Unifié

La couche de fédération devient le point de contrôle central pour la sécurité d'accès aux données. Une stratégie cohérente doit couvrir l'authentification, l'autorisation et l'audit.

Architecture de sécurité :



```
┌──────────────────────────────────────────────────────────────────────┐
│                    AUDIT                                             │
│  ┌──────────────────────────────────────────────────────────────┐    │
│  │  • Qui a accédé à quelles données                            │    │
│  │  • Quand et depuis où                                        │    │
│  │  • Quelles requêtes exécutées                                │    │
│  │  • Combien de données retournées                             │    │
│  └──────────────────────────────────────────────────────────────┘    │
└──────────────────────────────────────────────────────────────────────┘
```

Configuration OAuth2 Trino :

```
# etc/config.properties
http-server.authentication.type=OAUTH2

http-server.authentication.oauth2.issuer=https://login.microsoftonline.com/${TENANT_ID}/
v2.0
http-server.authentication.oauth2.client-id=${CLIENT_ID}
http-server.authentication.oauth2.client-secret=${CLIENT_SECRET}
http-server.authentication.oauth2.scopes=openid,profile,email
http-server.authentication.oauth2.principal-field=email
http-server.authentication.oauth2.groups-field=groups
```

Contrôle d'accès basé sur les rôles :

```
-- Création des rôles
CREATE ROLE data_analyst;
CREATE ROLE data_engineer;
CREATE ROLE data_scientist;
CREATE ROLE finance_analyst;

-- Permissions globales
GRANT USAGE ON CATALOG lakehouse TO ROLE data_analyst;
GRANT SELECT ON SCHEMA lakehouse.gold TO ROLE data_analyst;

GRANT USAGE ON CATALOG lakehouse TO ROLE data_engineer;
GRANT ALL PRIVILEGES ON SCHEMA lakehouse.bronze TO ROLE data_engineer;
GRANT ALL PRIVILEGES ON SCHEMA lakehouse.silver TO ROLE data_engineer;
GRANT SELECT ON SCHEMA lakehouse.gold TO ROLE data_engineer;

-- Permissions spécifiques au domaine Finance
GRANT SELECT ON TABLE lakehouse.gold.transactions TO ROLE finance_analyst;
GRANT SELECT ON TABLE lakehouse.gold.revenus_mensuels TO ROLE finance_analyst;

-- Attribution des rôles
GRANT ROLE data_analyst TO USER 'marie.tremblay@entreprise.ca';
GRANT ROLE finance_analyst TO GROUP 'Finance-Equipe@entreprise.ca';
```

Masquage et Filtrage des Données

Masquage au niveau colonne :

```
-- Fonction de masquage pour courriel
CREATE FUNCTION masque_courriel(val VARCHAR)
RETURNS VARCHAR
```

```
RETURN
    CASE
        WHEN current_user IN ('admin@entreprise.ca') THEN val
        WHEN is_role_enabled('data_admin') THEN val
        ELSE regexp_replace(val, '(.{2}).*@(.*)\.(.{2,3})$', '$1***@$2.$3')
    END;

-- Application du masquage
ALTER TABLE lakehouse.silver.clients
ALTER COLUMN courriel SET MASK masque_courriel;

-- Masquage pour numéro de carte
CREATE FUNCTION masque_carte(val VARCHAR)
RETURNS VARCHAR
RETURN
    CASE
        WHEN is_role_enabled('finance_admin') THEN val
        ELSE concat('****-****-****-', right(val, 4))
    END;
```

Filtrage au niveau ligne (Row-Level Security) :

```
-- Politique de filtrage par région
CREATE ROW ACCESS POLICY filtre_region ON lakehouse.gold.transactions
FOR SELECT
TO ROLE regional_analyst
USING (
    region IN (
        SELECT region_autorisee
        FROM lakehouse.security.autorisations_regions
        WHERE utilisateur = current_user
    )
);

-- Politique de filtrage par date (données récentes uniquement)
CREATE ROW ACCESS POLICY filtre_historique ON lakehouse.gold.transactions
FOR SELECT
TO ROLE analyst_junior
USING (
    date_transaction >= CURRENT_DATE - INTERVAL '90' DAY
);
```

Audit et Traçabilité

Configuration de l'audit Trino :

```
# etc/event-listener.properties
event-listener.name=http
http-client.request-timeout=30s
http-event-listener.connect-ingest-uri=https://audit.entreprise.ca/ingest
```

```
// Plugin d'audit personnalisé
public class AuditEventListener implements EventListener {
    @Override
```

```
    public void queryCompleted(QueryCompletedEvent event) {
        AuditRecord record = AuditRecord.builder()
            .queryId(event.getMetadata().getQueryId())
            .user(event.getContext().getUser())
            .query(event.getMetadata().getQuery())
            .catalog(event.getContext().getCatalog().orElse(""))
            .schema(event.getContext().getSchema().orElse(""))
            .tablesAccessed(extractTables(event))
            .rowsReturned(event.getStatistics().getOutputRows())
            .executionTimeMs(event.getStatistics().getWallTime().toMillis())
            .status(event.getMetadata().getQueryState())
            .timestamp(Instant.now())
            .build();

        auditService.log(record);
    }
}
```

Requêtes d'analyse d'audit :

```
-- Accès aux données sensibles (dernières 24h)
SELECT
    user_email,
    table_name,
    COUNT(*) as nb_requetes,
    SUM(rows_returned) as total_lignes
FROM audit.query_logs
WHERE timestamp >= CURRENT_TIMESTAMP - INTERVAL '24' HOUR
  AND table_name LIKE '%pii%' OR table_name LIKE '%sensitive%'
GROUP BY user_email, table_name
ORDER BY total_lignes DESC;

-- Détection d'anomalies (volume inhabituel)
WITH stats_utilisateur AS (
    SELECT
        user_email,
        AVG(rows_returned) as moy_lignes,
        STDDEV(rows_returned) as stddev_lignes
    FROM audit.query_logs
    WHERE timestamp >= CURRENT_TIMESTAMP - INTERVAL '30' DAY
    GROUP BY user_email
)
SELECT
    q.user_email,
    q.query_id,
    q.rows_returned,
    s.moy_lignes,
    (q.rows_returned - s.moy_lignes) / NULLIF(s.stddev_lignes, 0) as z_score
FROM audit.query_logs q
JOIN stats_utilisateur s ON q.user_email = s.user_email
WHERE q.timestamp >= CURRENT_TIMESTAMP - INTERVAL '1' HOUR
  AND (q.rows_returned - s.moy_lignes) / NULLIF(s.stddev_lignes, 0) > 3;
```

Intégration avec les Outils BI

Power BI et Microsoft Fabric

L'intégration de Power BI avec un Lakehouse Iceberg offre plusieurs chemins, chacun avec ses compromis.

Option 1 : Connexion directe via Trino/Dremio ODBC

```
Power BI Desktop → ODBC Driver → Trino/Dremio → Iceberg
```

Configuration du DSN ODBC :

```
# odbc.ini
[Trino_Lakehouse]
Driver=/opt/trino-odbc/lib/libtrino-odbc.so
Host=trino.lakehouse.entreprise.ca
Port=443
SSL=1
AuthenticationType=OAuth2
OAuth2ClientId=${CLIENT_ID}
OAuth2ClientSecret=${CLIENT_SECRET}
OAuth2TokenEndpoint=https://login.microsoftonline.com/${TENANT_ID}/oauth2/v2.0/token
Catalog=lakehouse
Schema=gold
```

Option 2 : Microsoft Fabric avec OneLake Shortcuts

```
-- Dans Fabric, création d'un raccourci vers S3
-- (Via interface Fabric ou API)
CREATE SHORTCUT gold_transactions
    LOCATION 's3://entreprise-lakehouse/gold/transactions'
    WITH (
        FORMAT = 'ICEBERG',
        CREDENTIAL = 'aws_credential'
    );
```

Option 3 : Direct Lake avec Dremio

Dremio peut exposer ses sources comme endpoints compatibles Power BI Direct Lake :

```
-- Configuration Dremio pour Power BI
ALTER SOURCE "Lakehouse Gold" SET (
    ENABLE_POWERBI_DIRECTLAKE = true,
    POWERBI_WORKSPACE_ID = '${WORKSPACE_ID}'
);
```

> **Performance**
>
> Direct Lake offre les meilleures performances pour Power BI car les données restent dans le format Parquet/Iceberg sans import. Les rapports interrogent directement le Lakehouse avec un cache optimisé. Pour les datasets volumineux (> 100 Go), Direct Lake peut réduire les temps de rafraîchissement de 90% par rapport à l'import.

Tableau

Connexion Tableau via JDBC :

```
<!-- tableau-trino.tdc -->
<connection-customization class='genericjdbc' enabled='true' version='1.0'>
    <vendor name='trino' />
    <driver name='trino' />
    <customizations>
        <customization name='CAP_QUERY_SUBQUERIES_WITH_TOP' value='no' />
        <customization name='CAP_SELECT_INTO' value='no' />
        <customization name='CAP_SUPPORTS_UNION' value='yes' />
        <customization name='SQL_AGGREGATE_FUNCTION_STDDEV' value='STDDEV_SAMP' />
        <customization name='SQL_AGGREGATE_FUNCTION_VAR' value='VAR_SAMP' />
    </customizations>
</connection-customization>
```

Live connection optimisée :

```
# Paramètres de connexion Tableau
Server: trino.lakehouse.entreprise.ca
Port: 443
Catalog: lakehouse
Schema: gold
Authentication: OAuth
Initial SQL: SET SESSION query_max_execution_time = '10m'
```

Superset et Metabase

Configuration Superset :

```
# superset_config.py
SQLALCHEMY_DATABASE_URI = 'trino://trino.lakehouse.entreprise.ca:443/lakehouse'

DATABASES = {
    'lakehouse': {
        'engine': 'trino',
        'host': 'trino.lakehouse.entreprise.ca',
        'port': 443,
        'catalog': 'lakehouse',
        'schema': 'gold',
        'extra': {
            'http_scheme': 'https',
            'auth': 'oauth2',
            'oauth2_config': {
                'client_id': '${CLIENT_ID}',
                'client_secret': '${CLIENT_SECRET}',
                'token_url': 'https://login.microsoftonline.com/${TENANT_ID}/oauth2/v2.0/
token'
            }
        }
    }
}
```

Configuration Metabase :

```
# Via API Metabase
POST /api/database
{
  "engine": "presto-jdbc",
  "name": "Lakehouse Analytics",
  "details": {
    "host": "trino.lakehouse.entreprise.ca",
    "port": 443,
    "catalog": "lakehouse",
    "schema": "gold",
    "ssl": true,
    "auth-method": "oauth2",
    "oauth2-client-id": "${CLIENT_ID}",
    "oauth2-client-secret": "${CLIENT_SECRET}"
  }
}
```

Études de Cas Canadiennes

Secteur Bancaire : Institution Financière Majeure

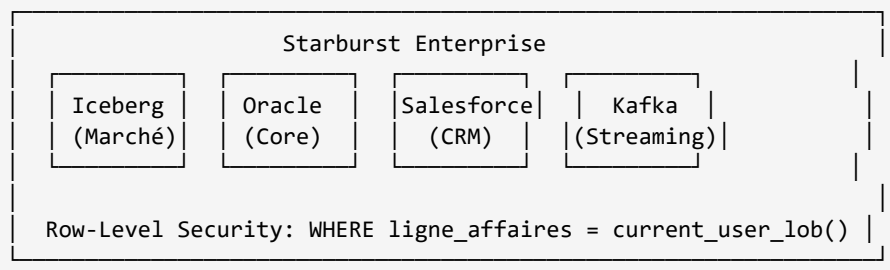
Étude de cas : Grande banque canadienne

Secteur : Services financiers

Défi : Unifier l'accès analytique à 15+ silos de données (core banking, CRM, risque, conformité) tout en maintenant une ségrégation stricte entre lignes d'affaires conformément aux exigences du BSIF. Les analystes passaient 60% de leur temps à chercher et préparer les données plutôt qu'à les analyser.

Solution : Déploiement de Starburst Enterprise comme couche de fédération, connectant des catalogues Iceberg (données de marché, transactions), des bases Oracle (core banking), et Salesforce (CRM). Implémentation de row-level security basée sur la ligne d'affaires de l'utilisateur.

Architecture :



Résultats :

- Temps de préparation des données réduit de 60% à 15%
- 500+ analystes avec accès self-service sécurisé

- Conformité BSIF maintenue avec audit complet
- ROI de 400% sur 18 mois

Secteur Commerce : Détaillant Omnicanal

Étude de cas : Chaîne de magasins pancanadienne

Secteur : Commerce de détail

Défi : Créer une vue unifiée des clients cross-canal (magasin, web, application mobile, marketplace) pour personnalisation en temps réel. Les données étaient dispersées entre 8 systèmes avec des identifiants clients incompatibles.

Solution : Déploiement de Dremio Cloud comme plateforme de fédération avec couche sémantique. Création d'un graphe d'identité client via Iceberg, exposé comme vue virtuelle « Client 360 ». Reflections pour accélération des tableaux de bord Power BI temps réel.

Particularités :

- Resolution d'identité via algorithme de matching probabiliste
- Rafraîchissement des Reflections toutes les 15 minutes
- Intégration avec système de recommandation ML

Résultats :

- Vue client unifiée couvrant 8M de clients
- Latence des tableaux de bord < 2 secondes (vs 45 secondes avant)
- Augmentation de 23% du taux de conversion des recommandations
- Réduction de 70% des coûts d'infrastructure BI

Secteur Santé : Réseau Hospitalier Provincial

Étude de cas : Réseau de santé provincial

Secteur : Santé publique

Défi : Permettre l'analyse populationnelle tout en respectant strictement la confidentialité des données de santé. Les chercheurs avaient besoin d'accéder à des données agrégées sans jamais voir de données individuelles identifiantes.

Solution : Architecture Trino avec couche de fédération hautement sécurisée. Implémentation de k-anonymisation automatique pour toutes les requêtes retournant moins de 5 individus. Masquage systématique des identifiants directs et quasi-identifiants.

Architecture de sécurité :

```
-- Politique de k-anonymisation
CREATE POLICY k_anon ON requete_resultat
WHEN COUNT(*) < 5 THEN
    SUPPRESS_RESULT('Résultat supprimé: k < 5')
```

Résultats :

- Conformité Loi 25 et LPRPDE validée par commissaire à la vie privée
- 200+ chercheurs avec accès sécurisé
- Temps d'accès aux données pour recherche réduit de 6 mois à 2 semaines
- Aucune brèche de confidentialité depuis le déploiement (3 ans)

Secteur Énergie : Producteur d'Électricité

Étude de cas : Société d'État hydroélectrique

Secteur : Énergie et services publics

Défi : Analyser en temps réel les données de 50 000 capteurs IoT répartis sur le réseau de production et distribution, combinées aux données météorologiques, de marché et de maintenance pour optimisation de la production.

Solution : Trino déployé sur Kubernetes avec auto-scaling pour gérer les pics de charge. Fédération entre Iceberg (historique capteurs), TimescaleDB (séries temporelles temps réel), et API météo externes. Cache Alluxio pour accélération des lectures répétitives.

Métriques :

- 50 milliards de points de données/jour ingérés
- Requêtes analytiques en < 10 secondes sur 2 ans d'historique
- 200 requêtes concurrentes en pic

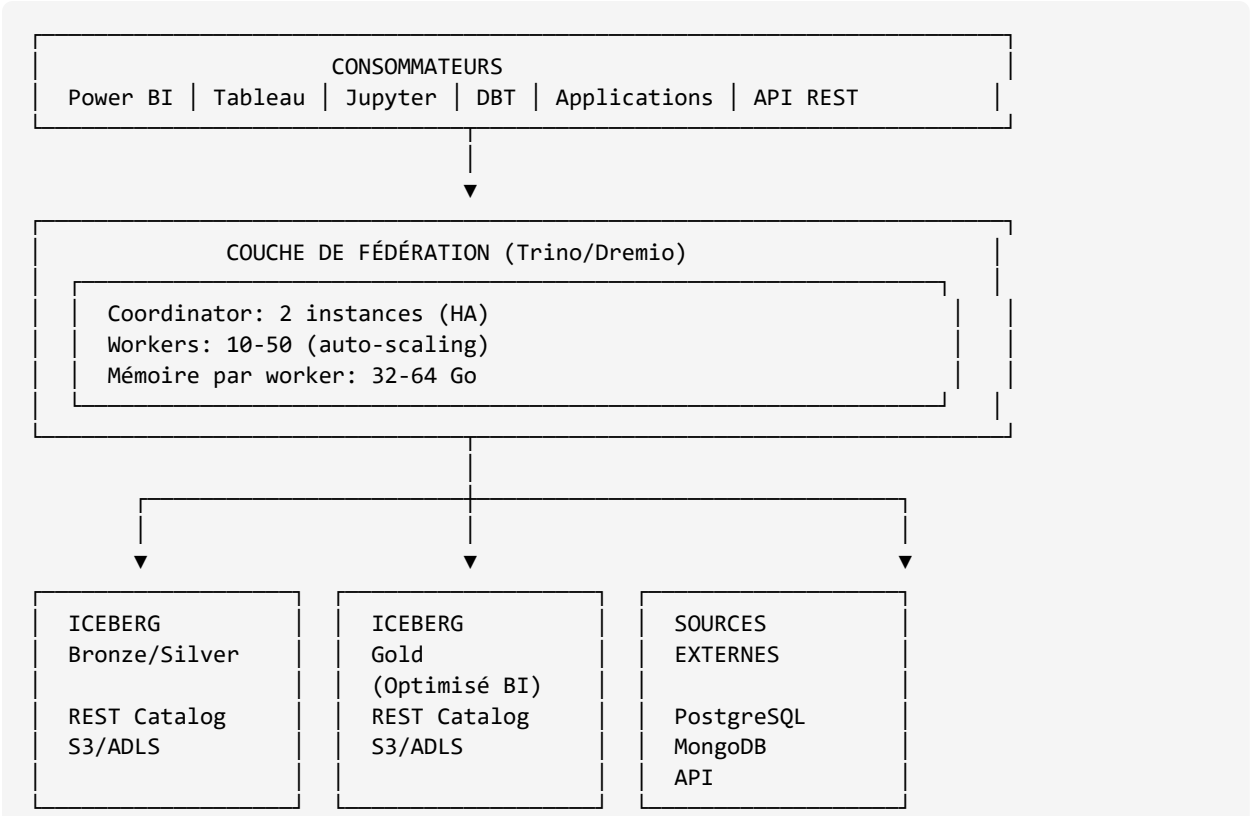
Résultats :

- Optimisation de la production augmentant les revenus de 3%
- Détection précoce des anomalies réduisant les pannes de 25%
- Économies de 15M$/an en maintenance prédictive

Bonnes Pratiques et Recommandations

Architecture de Référence

Architecture recommandée pour entreprise moyenne :



Checklist de Déploiement

Infrastructure :

- ☐ Cluster Trino/Dremio dimensionné selon charge attendue
- ☐ Haute disponibilité du coordinator configurée
- ☐ Auto-scaling des workers activé
- ☐ Monitoring et alerting en place
- ☐ Sauvegarde des configurations

Sécurité :

- ☐ Authentification OAuth2/OIDC configurée
- ☐ Rôles et permissions définis
- ☐ Masquage des données sensibles implémenté
- ☐ Row-level security si nécessaire
- ☐ Audit activé et journaux centralisés

Performance :

- ☐ Predicate pushdown vérifié pour chaque connecteur
- ☐ Cache de métadonnées configuré
- ☐ Reflections/matérialisations pour requêtes fréquentes
- ☐ Statistiques de table à jour

Gouvernance :

- ☐ Documentation des sources et vues
- ☐ Lignage des données tracé

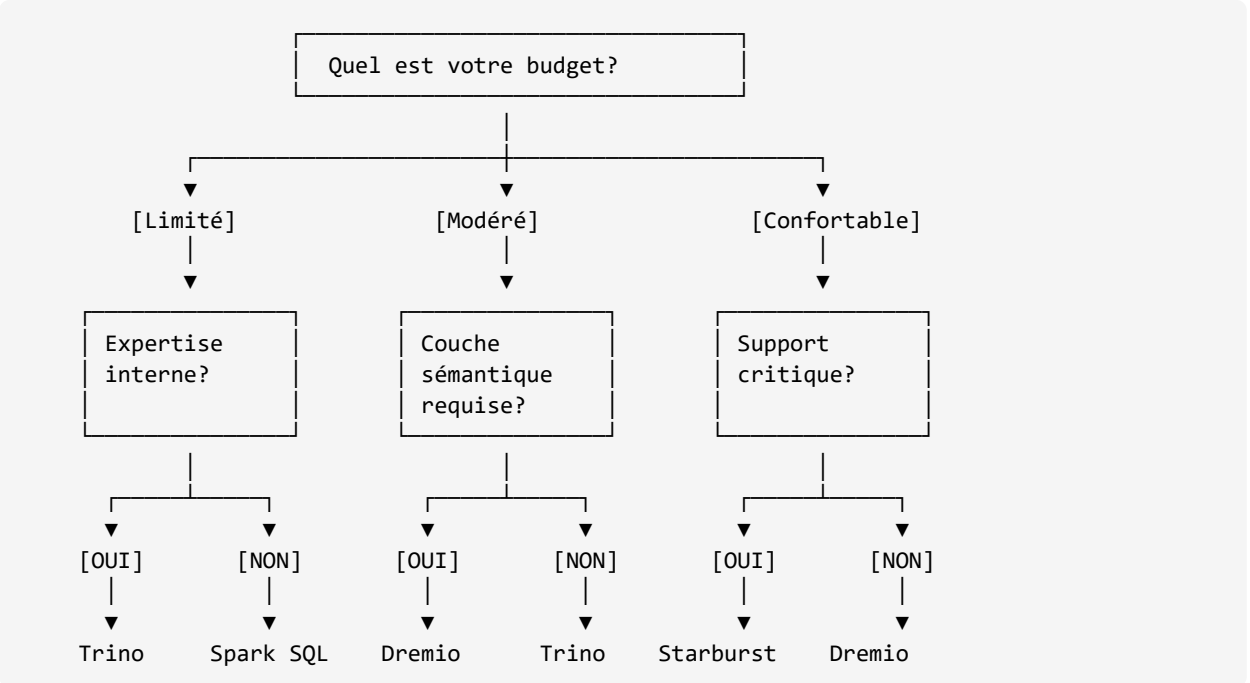
- ☐ Glossaire métier défini
- ☐ Propriétaires de données identifiés

Matrice de Décision

Choix du moteur de fédération :

Critère	Trino	Dremio	Spark SQL	Starburst
Budget limité	★★★★★	★★☆☆☆	★★★★★	★☆☆☆☆
Requêtes interactives	★★★★★	★★★★★	★★☆☆☆	★★★★★
Couche sémantique	★★☆☆☆	★★★★★	★★☆☆☆	★★★☆☆
Simplicité déploiement	★★★☆☆	★★★★★	★★☆☆☆	★★★★☆
Connecteurs sources	★★★★★	★★★★☆	★★★☆☆	★★★★★
Support entreprise	★★☆☆☆	★★★★★	★★★☆☆	★★★★★
Intégration BI	★★★★☆	★★★★★	★★★☆☆	★★★★☆

Arbre de décision :



Conclusion

La couche de fédération transforme votre Data Lakehouse Apache Iceberg d'une infrastructure de stockage en une plateforme analytique accessible à l'ensemble de l'organisation. Elle unifie l'accès à des sources hétérogènes, optimise l'exécution des requêtes et centralise la gouvernance des données. Le choix du moteur de fédération — Trino pour sa flexibilité open source, Dremio pour sa couche sémantique

intégrée, ou Starburst pour le support entreprise — dépend de votre contexte spécifique en termes de budget, d'expertise et d'exigences fonctionnelles.

Les performances de cette couche reposent sur une configuration soignée : predicate et projection pushdown correctement activés, cache de métadonnées dimensionné, stratégies de jointure optimisées et, pour les cas d'usage répétitifs, matérialisations via Reflections ou vues matérialisées. Le dimensionnement doit anticiper les pics de charge avec des mécanismes d'auto-scaling appropriés.

La sécurité et la gouvernance constituent des piliers incontournables. L'authentification moderne via OAuth2/OIDC, le contrôle d'accès granulaire jusqu'au niveau colonne et ligne, le masquage des données sensibles et l'audit exhaustif permettent de démocratiser l'accès aux données tout en maintenant la conformité réglementaire — exigence particulièrement critique dans le contexte canadien avec la Loi 25 et les régulations sectorielles.

Les études de cas présentées démontrent que des organisations de secteurs variés — finance, commerce, santé, énergie — ont réussi à déployer ces architectures de fédération pour transformer leurs capacités analytiques. Les gains mesurés sont significatifs : réduction drastique du temps de préparation des données, accélération des temps de réponse, et amélioration mesurable des décisions métier.

Le chapitre suivant explore la couche de consommation et les patterns d'accès aux données, où nous examinerons comment les différents profils d'utilisateurs — analystes, scientifiques de données, applications — interagissent avec votre Lakehouse fédéré pour extraire de la valeur des données.

Résumé

Fondamentaux de la fédération :

- Accès unifié à des sources hétérogènes via SQL standard
- Réduction de la duplication par requêtes in situ
- Point de contrôle central pour sécurité et gouvernance
- Patterns : local Iceberg, inter-catalogue, hybride, virtualisation

Moteurs principaux :

- **Trino** : Standard open source, MPP, 40+ connecteurs, gratuit
- **Dremio** : Couche sémantique, Reflections, interface intuitive
- **Spark SQL** : Intégration pipelines, requêtes longues, ML
- **Starburst** : Trino entreprise, support, gouvernance avancée

Optimisations de performance :

- Predicate et projection pushdown essentiels
- Stratégies de jointure : broadcast, hash, merge selon tailles
- Cache métadonnées et Reflections pour requêtes répétitives
- Auto-scaling pour gérer la variabilité de charge

Sécurité et gouvernance :

- Authentification OAuth2/OIDC avec IdP entreprise
- RBAC au niveau catalogue, schema, table, colonne, ligne
- Masquage des données sensibles via fonctions
- Audit exhaustif pour traçabilité et conformité

Intégration BI :

- Power BI : ODBC, Fabric shortcuts, Direct Lake
- Tableau : JDBC avec customizations
- Superset/Metabase : configurations natives

Recommandations :

- Trino pour budget limité et flexibilité
- Dremio pour couche sémantique et simplicité
- Starburst pour support entreprise critique
- Toujours activer pushdown et configurer le cache

Ce chapitre établit l'interface d'accès à votre Lakehouse. Le chapitre suivant, « Conception de la Couche de Consommation », détaille les patterns d'accès pour les différents profils d'utilisateurs et applications.

Chapitre IV.9 - Comprendre la Couche de Consommation

Introduction

La couche de consommation représente l'aboutissement de votre architecture Data Lakehouse — le moment où les données méticuleusement collectées, transformées et gouvernées deviennent valeur tangible pour l'organisation. Elle définit comment les différents profils d'utilisateurs et d'applications accèdent aux données, les interrogent et les exploitent pour prendre des décisions éclairées. Une couche de consommation bien conçue démocratise l'accès aux données tout en préservant la sécurité et la performance ; mal conçue, elle crée frustration, goulots d'étranglement et risques de gouvernance.

Dans l'écosystème Apache Iceberg, la consommation des données bénéficie de caractéristiques uniques qui transforment l'expérience utilisateur. Le Time Travel permet aux analystes d'explorer les états historiques des données sans intervention technique. L'évolution de schéma garantit que les requêtes existantes continuent de fonctionner malgré les modifications structurelles. Le partitionnement masqué libère les utilisateurs de la complexité technique du stockage. Ces fonctionnalités, invisibles pour la plupart des consommateurs, élèvent considérablement la qualité de l'expérience analytique.

Ce chapitre explore en profondeur les patterns de consommation adaptés à chaque profil d'utilisateur : analystes d'affaires explorant les données via Power BI ou Tableau, scientifiques de données entraînant des modèles depuis leurs notebooks Jupyter, applications métier interrogeant le Lakehouse via API, et ingénieurs de données construisant des pipelines automatisés. Pour chaque profil, nous examinerons les outils appropriés, les configurations optimales et les bonnes pratiques garantissant performance et gouvernance.

L'enjeu dépasse la simple connectivité technique. La couche de consommation façonne la culture de données de votre organisation. Elle détermine qui peut accéder à quelles informations, avec quelle facilité et quelle rapidité. Une démocratisation réussie des données — où chaque collaborateur dispose des informations nécessaires à ses décisions — repose sur une architecture de consommation inclusive, performante et sécurisée.

Taxonomie des Consommateurs de Données

Profils d'Utilisateurs

La diversité des consommateurs de données nécessite une compréhension fine de leurs besoins, compétences et patterns d'utilisation distincts. Cette taxonomie guide les choix architecturaux et les priorités d'optimisation.

Analystes d'affaires : Utilisateurs orientés métier, ils explorent les données pour répondre à des questions business. Leur expertise réside dans la compréhension du domaine plutôt que dans les technologies de données. Ils privilégient les interfaces visuelles, les requêtes ad-hoc et les tableaux de bord interactifs.

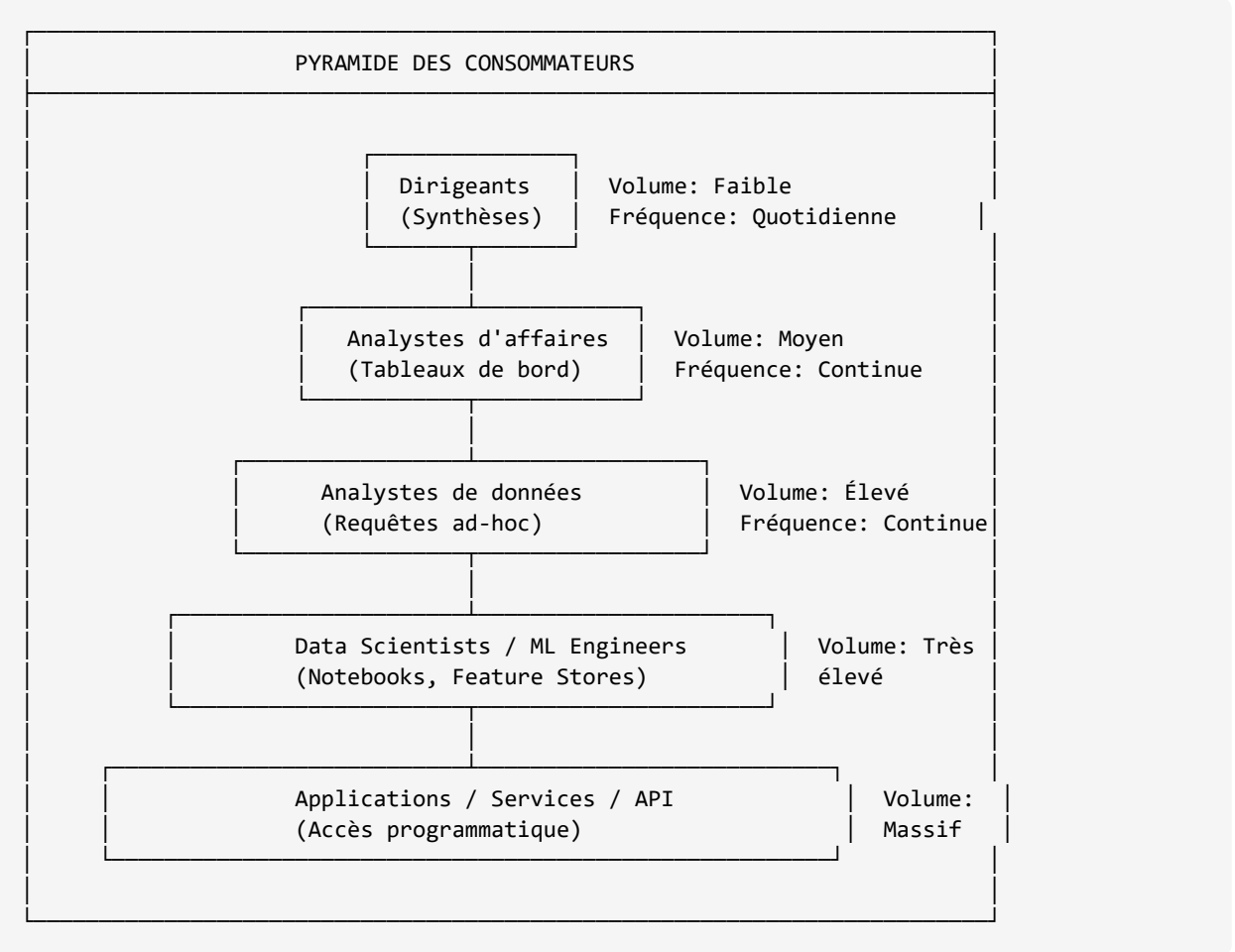
Analystes de données : À la frontière entre métier et technique, ils maîtrisent SQL et les outils de visualisation avancés. Ils construisent des rapports complexes, effectuent des analyses exploratoires et préparent des datasets pour d'autres consommateurs.

Scientifiques de données : Experts en statistiques et apprentissage automatique, ils accèdent aux données via notebooks et frameworks ML. Leurs besoins incluent l'accès à de grands volumes de données brutes, la reproductibilité des expériences et l'intégration avec les pipelines MLOps.

Ingénieurs de données : Constructeurs des pipelines de données, ils accèdent programmatiquement au Lakehouse pour l'ingestion, la transformation et l'orchestration. Leur focus est la fiabilité, la performance et l'automatisation.

Applications et services : Systèmes automatisés consommant les données via API pour alimenter des fonctionnalités métier — personnalisation, détection de fraude, tableaux de bord embarqués.

Dirigeants et décideurs : Consommateurs de synthèses et indicateurs clés, ils accèdent aux données via des rapports consolidés et des alertes automatisées plutôt que par exploration directe.



Patterns d'Accès Caractéristiques

Chaque profil d'utilisateur exhibe des patterns d'accès distincts qui influencent les choix d'optimisation.

Profil	Pattern dominant	Volume typique	Latence attendue	Fréquence
Dirigeants	Lecture KPI agrégés	Ko-Mo	< 1s	Quotidienne
Analystes affaires	Requêtes filtrees, drill-down	Mo-Go	< 5s	Continue
Analystes données	Scans larges, jointures	Go-To	< 30s	Continue
Data Scientists	Scans complets, échantillonnage	To	Minutes	Par projet
Applications	Lookups par clé, agrégats	Ko-Mo	< 100ms	Temps réel
Pipelines	Lectures/écritures massives	To	Minutes-Heures	Planifiée

Exigences par Profil

Exigences fonctionnelles :

Profil	Interface préférée	Fonctionnalités clés
Analystes affaires	GUI (Power BI, Tableau)	Visualisation, filtres, export
Analystes données	SQL + GUI	Requêtes complexes, joins, CTEs
Data Scientists	Notebooks + DataFrames	Accès brut, échantillonnage, ML
Applications	API REST/gRPC	Faible latence, haute disponibilité
Pipelines	SDK programmatique	Transactions, streaming, batch

Exigences non-fonctionnelles :

Profil	Priorité performance	Priorité disponibilité	Tolérance complexité
Dirigeants	Élevée	Élevée	Aucune
Analystes affaires	Élevée	Moyenne	Faible
Analystes données	Moyenne	Moyenne	Moyenne
Data Scientists	Moyenne	Moyenne	Élevée
Applications	Critique	Critique	Élevée
Pipelines	Variable	Élevée	Élevée

Consommation pour l'Intelligence d'Affaires

Intégration Power BI

Microsoft Power BI représente l'outil de BI dominant dans de nombreuses organisations canadiennes, particulièrement celles investies dans l'écosystème Microsoft. L'intégration avec un Lakehouse Iceberg offre plusieurs chemins, chacun avec ses compromis.

Mode Import : Les données sont copiées dans le modèle Power BI.

```
Lakehouse Iceberg → Requête → Import Power BI → Modèle en mémoire
```

Avantages :

- Performance de requête optimale (données en mémoire)
- Fonctionnement hors ligne
- Compression et optimisation automatiques

Limitations :

- Données non temps réel (rafraîchissement planifié)
- Limite de taille du modèle (selon licence)
- Duplication des données

Mode DirectQuery : Les requêtes sont exécutées directement sur le Lakehouse.

```
Utilisateur → Power BI → DirectQuery → Trino/Dremio → Iceberg
```

Avantages :

- Données toujours à jour
- Pas de limite de volume
- Pas de duplication

Limitations :

- Performance dépendante du Lakehouse
- Fonctionnalités DAX limitées
- Charge sur l'infrastructure

Mode Direct Lake (Microsoft Fabric) : Lecture directe des fichiers Parquet/Iceberg sans import.

```
Utilisateur → Power BI → Direct Lake → OneLake → Iceberg (via shortcut)
```

Avantages :

- Performance proche de l'import
- Données quasi temps réel
- Pas de duplication

Limitations :

- Requiert Microsoft Fabric
- Configuration OneLake nécessaire

Configuration Power BI avec Trino :

```
# Chaîne de connexion ODBC
Driver={Trino ODBC Driver};
Host=trino.lakehouse.entreprise.ca;
Port=443;
Catalog=lakehouse;
Schema=gold;
SSL=1;
AuthenticationType=OAuth2;
OAuth2ClientId=<CLIENT_ID>;
OAuth2ClientSecret=<CLIENT_SECRET>;
OAuth2TokenEndpoint=https://login.microsoftonline.com/<TENANT>/oauth2/v2.0/token;
```

Optimisation des rapports Power BI :

```
-- Vue optimisée pour tableau de bord exécutif
CREATE VIEW lakehouse.semantic.kpi_ventes_quotidien AS
SELECT
    date_transaction,
    region,
    categorie_produit,
    SUM(montant) as chiffre_affaires,
    COUNT(DISTINCT client_id) as clients_uniques,
    COUNT(*) as nombre_transactions,
    AVG(montant) as panier_moyen
FROM lakehouse.gold.transactions
WHERE date_transaction >= CURRENT_DATE - INTERVAL '365' DAY
GROUP BY date_transaction, region, categorie_produit;

-- Statistiques pour optimisation des requêtes
ANALYZE lakehouse.semantic.kpi_ventes_quotidien;
```

Intégration Tableau

Tableau offre une expérience d'exploration visuelle puissante avec plusieurs options de connexion au Lakehouse Iceberg.

Connexion Tableau via JDBC :

```
<!-- tableau-connection.tdc -->
<connection-customization class='genericjdbc' enabled='true'>
    <vendor name='trino' />
    <driver name='trino' />
    <customizations>
        <customization name='CAP_CREATE_TEMP_TABLES' value='yes' />
        <customization name='CAP_QUERY_BOOLEXPR_TO_INTEXPR' value='no' />
        <customization name='CAP_QUERY_GROUP_BY_ALIAS' value='yes' />
        <customization name='CAP_QUERY_SUBQUERIES' value='yes' />
        <customization name='CAP_QUERY_SUBQUERIES_WITH_TOP' value='yes' />
        <customization name='SQL_AGGREGATE_FUNCTION_STDDEV' value='STDDEV_SAMP' />
        <customization name='SQL_AGGREGATE_FUNCTION_VAR' value='VAR_SAMP' />
    </customizations>
</connection-customization>
```

Extraits Tableau optimisés :

Pour les analyses répétitives, les extraits Tableau (.hyper) offrent des performances supérieures :

```
-- Requête source pour extrait
SELECT
    t.transaction_id,
    t.date_transaction,
    t.montant,
    t.quantite,
    c.segment_client,
    c.region,
    p.categorie,
    p.sous_categorie
FROM lakehouse.gold.transactions t
JOIN lakehouse.gold.dim_clients c ON t.client_id = c.client_id
JOIN lakehouse.gold.dim_produits p ON t.produit_id = p.produit_id
WHERE t.date_transaction >= DATE_ADD('year', -2, CURRENT_DATE)
```

Tableau Server avec connexions publiées :

```
# Configuration datasource Tableau Server
datasource:
  name: "Lakehouse - Ventes Gold"
  connection:
    class: "genericjdbc"
    dbname: "lakehouse"
    schema: "gold"
    server: "trino.lakehouse.entreprise.ca"
    port: "443"
    authentication: "oauth"
    oauth-config:
      client-id: "${TABLEAU_CLIENT_ID}"
      auth-uri: "https://login.microsoftonline.com/${TENANT}/oauth2/v2.0/authorize"
      token-uri: "https://login.microsoftonline.com/${TENANT}/oauth2/v2.0/token"
  refresh-schedule:
    frequency: "daily"
    time: "06:00"
    timezone: "America/Toronto"
```

Self-Service Analytics

L'analytique en libre-service permet aux utilisateurs métier d'explorer les données sans dépendance constante envers les équipes techniques. Cette démocratisation repose sur une préparation soignée de la couche sémantique.

Principes de conception self-service :

1. **Tables larges dénormalisées** : Réduire les jointures pour les utilisateurs non techniques
2. **Nommage métier** : Utiliser des noms compréhensibles (`chiffre_affaires` plutôt que `amt_ttc`)
3. **Documentation intégrée** : Descriptions et exemples dans les métadonnées
4. **Métriques pré-calculées** : Éviter les calculs complexes côté utilisateur
5. **Filtres par défaut** : Limiter automatiquement aux données pertinentes

Création d'une couche sémantique self-service :

```
-- Table analytique optimisée pour self-service
CREATE TABLE lakehouse.semantic.analyse_ventes (
    -- Dimensions temporelles
    date_transaction DATE COMMENT 'Date de la transaction',
    annee INT COMMENT 'Année (ex: 2024)',
    trimestre VARCHAR COMMENT 'Trimestre (ex: T1 2024)',
    mois VARCHAR COMMENT 'Mois (ex: Janvier 2024)',
    semaine_annee INT COMMENT 'Numéro de semaine dans l''année',
    jour_semaine VARCHAR COMMENT 'Jour de la semaine (ex: Lundi)',
    est_fin_semaine BOOLEAN COMMENT 'Vrai si samedi ou dimanche',

    -- Dimensions client
    client_id BIGINT COMMENT 'Identifiant unique du client',
    segment_client VARCHAR COMMENT 'Segment marketing (Premium, Standard, Économique)',
    ancienneté_client VARCHAR COMMENT 'Ancienneté (Nouveau, 1-2 ans, 3-5 ans, 5+ ans)',
    province_client VARCHAR COMMENT 'Province de résidence',
    ville_client VARCHAR COMMENT 'Ville de résidence',

    -- Dimensions produit
    categorie_produit VARCHAR COMMENT 'Catégorie principale du produit',
    sous_categorie VARCHAR COMMENT 'Sous-catégorie du produit',
    marque VARCHAR COMMENT 'Marque du produit',

    -- Dimensions géographiques
    region_vente VARCHAR COMMENT 'Région de vente',
    magasin VARCHAR COMMENT 'Nom du magasin ou "En ligne"',
    canal VARCHAR COMMENT 'Canal de vente (Magasin, Web, Mobile)',

    -- Métriques
    chiffre_affaires DECIMAL(15,2) COMMENT 'Montant total de la vente en CAD',
    quantite INT COMMENT 'Nombre d''unités vendues',
    marge_brute DECIMAL(15,2) COMMENT 'Marge brute en CAD',
    taux_marge DECIMAL(5,2) COMMENT 'Taux de marge en pourcentage',
    cout_acquisition DECIMAL(10,2) COMMENT 'Coût d''acquisition attribué'
)
USING iceberg
PARTITIONED BY (annee, trimestre)
COMMENT 'Table analytique des ventes pour exploration self-service.
Mise à jour quotidienne à 6h00. Données disponibles depuis 2020.';

-- Ajout de propriétés pour documentation
ALTER TABLE lakehouse.semantic.analyse_ventes SET TBLPROPERTIES (
    'owner' = 'equipe-analytics@entreprise.ca',
    'data-steward' = 'marie.tremblay@entreprise.ca',
    'refresh-frequency' = 'daily',
    'last-refresh' = '2024-01-15T06:00:00Z',
    'row-count' = '45000000',
    'documentation-url' = 'https://wiki.entreprise.ca/data/ventes'
);
```

Performance

Les tables self-service dénormalisées augmentent le volume de stockage (redondance) mais réduisent drastiquement la complexité des requêtes utilisateur et améliorent les performances. Pour une table de 50 millions de transactions, la dénormalisation typique augmente le stockage de 30-50% mais réduit le temps de requête moyen de 60-80% en éliminant les jointures.

Consommation pour la Science des Données

Accès Notebook et DataFrame

Les scientifiques de données privilégient l'accès programmatique via notebooks (Jupyter, Databricks, Google Colab) et frameworks DataFrame (pandas, Polars, PySpark). L'intégration avec Iceberg doit supporter ces workflows tout en garantissant la reproductibilité.

Configuration PyIceberg pour notebooks :

```
# Installation
# pip install pyiceberg[s3,pyarrow]

from pyiceberg.catalog import load_catalog
import pyarrow as pa

# Configuration du catalogue
catalog = load_catalog(
    "lakehouse",
    **{
        "type": "rest",
        "uri": "https://catalog.lakehouse.entreprise.ca",
        "credential": os.environ["ICEBERG_TOKEN"],
        "warehouse": "s3://entreprise-lakehouse/warehouse",
        "s3.region": "ca-central-1"
    }
)

# Chargement d'une table
table = catalog.load_table("gold.transactions")

# Conversion en DataFrame Arrow (efficace pour grandes tables)
arrow_table = table.scan(
    row_filter="date_transaction >= '2024-01-01'",
    selected_fields=["transaction_id", "client_id", "montant", "date_transaction"]
).to_arrow()

# Conversion en pandas
df = arrow_table.to_pandas()

# Ou directement en Polars (plus performant pour grandes tables)
import polars as pl
df_polars = pl.from_arrow(arrow_table)
```

Accès PySpark pour volumes importants :

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("DataScienceWorkbench") \
    .config("spark.sql.extensions",
            "org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions") \
    .config("spark.sql.catalog.lakehouse",
```

```
            "org.apache.iceberg.spark.SparkCatalog") \
    .config("spark.sql.catalog.lakehouse.type", "rest") \
    .config("spark.sql.catalog.lakehouse.uri",
            "https://catalog.lakehouse.entreprise.ca") \
    .config("spark.sql.catalog.lakehouse.warehouse",
            "s3://entreprise-lakehouse/warehouse") \
    .getOrCreate()

# Lecture avec filtrage pushdown
df_train = spark.read.table("lakehouse.gold.transactions") \
    .filter("date_transaction >= '2023-01-01'") \
    .filter("date_transaction < '2024-01-01'")

# Échantillonnage pour exploration
df_sample = df_train.sample(fraction=0.01, seed=42)

# Conversion en pandas pour ML local
pdf = df_sample.toPandas()
```

Reproductibilité avec Time Travel :

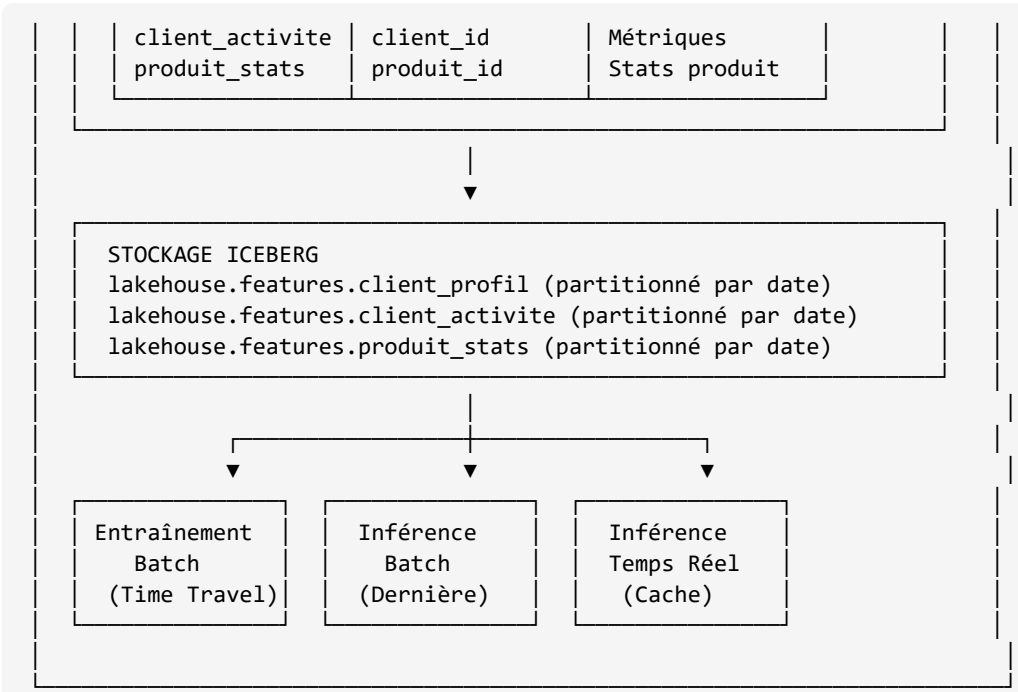
```
# Lecture d'un snapshot spécifique pour reproductibilité
snapshot_id = 1234567890123456789

# Via PyIceberg
table = catalog.load_table("gold.transactions")
arrow_table = table.scan(snapshot_id=snapshot_id).to_arrow()

# Via Spark
df_reproducible = spark.read \
    .option("snapshot-id", snapshot_id) \
    .table("lakehouse.gold.transactions")

# Ou via timestamp
df_at_time = spark.read \
    .option("as-of-timestamp", "2024-01-01T00:00:00Z") \
    .table("lakehouse.gold.transactions")
```

Feature Stores et Iceberg

Les Feature Stores centralisent la gestion des features ML, garantissant cohérence entre entraînement et inférence. Apache Iceberg s'intègre naturellement comme backend de stockage pour ces systèmes.

Architecture Feature Store avec Iceberg :

```
┌─────────────────────────────────────────────────────────────────┐
│                        FEATURE STORE                            │
├─────────────────────────────────────────────────────────────────┤
│                                                                 │
│  ┌─────────────────────────────────────────────────────────┐    │
│  │  REGISTRE DES FEATURES                                  │    │
│  │  ┌──────────────────┬──────────────────┬──────────────┐ │    │
│  │  │ Feature Group    │ Entity           │ Description  │ │    │
│  │  ├──────────────────┼──────────────────┼──────────────┤ │    │
│  │  │ client_profil    │ client_id        │ Profil client│ │    │
```



Implémentation avec Feast :

```yaml
# feature_store.yaml
project: lakehouse_ml
registry: s3://entreprise-lakehouse/feast/registry.db
provider: local
offline_store:
  type: spark
  spark_conf:
    spark.sql.catalog.lakehouse: org.apache.iceberg.spark.SparkCatalog
    spark.sql.catalog.lakehouse.type: rest
    spark.sql.catalog.lakehouse.uri: https://catalog.lakehouse.entreprise.ca
online_store:
  type: redis
  connection_string: redis://redis-features:6379
```

```python
# features.py
from feast import Entity, Feature, FeatureView, FileSource
from feast.types import Float32, Int64
from datetime import timedelta

# Entité client
client = Entity(
    name="client_id",
    description="Identifiant unique du client"
)

# Source Iceberg pour features client
client_features_source = FileSource(
    name="client_features_source",
    path="iceberg://lakehouse.features.client_profil",
    timestamp_field="date_feature"
)
```

```
# Feature View
client_features = FeatureView(
    name="client_features",
    entities=[client],
    ttl=timedelta(days=1),
    schema=[
        Feature(name="total_achats_30j", dtype=Float32),
        Feature(name="nb_transactions_30j", dtype=Int64),
        Feature(name="panier_moyen_30j", dtype=Float32),
        Feature(name="jours_depuis_dernier_achat", dtype=Int64),
        Feature(name="score_fidelite", dtype=Float32),
    ],
    source=client_features_source,
    online=True
)
```

Lecture des features pour entraînement :

```
from feast import FeatureStore
import pandas as pd

store = FeatureStore(repo_path=".")

# Entités pour lesquelles récupérer les features
entity_df = pd.DataFrame({
    "client_id": [1001, 1002, 1003, 1004, 1005],
    "event_timestamp": pd.to_datetime(["2024-01-15"] * 5)
})

# Récupération des features (utilise Time Travel Iceberg)
training_df = store.get_historical_features(
    entity_df=entity_df,
    features=[
        "client_features:total_achats_30j",
        "client_features:nb_transactions_30j",
        "client_features:panier_moyen_30j",
        "client_features:score_fidelite"
    ]
).to_df()
```

MLOps et Intégration Continue

L'intégration d'Iceberg dans les pipelines MLOps garantit la reproductibilité, la traçabilité et l'automatisation des workflows de machine learning.

Pipeline MLOps avec versionnement Iceberg :

```
import mlflow
from mlflow.tracking import MlflowClient

# Configuration MLflow
mlflow.set_tracking_uri("https://mlflow.lakehouse.entreprise.ca")
mlflow.set_experiment("prediction_churn")
```

```
def train_model(snapshot_id: int):
    """Entraînement reproductible avec snapshot Iceberg."""

    with mlflow.start_run():
        # Logging du snapshot utilisé pour reproductibilité
        mlflow.log_param("iceberg_snapshot_id", snapshot_id)
        mlflow.log_param("iceberg_table", "lakehouse.gold.client_features")

        # Chargement des données depuis snapshot spécifique
        df_train = spark.read \
            .option("snapshot-id", snapshot_id) \
            .table("lakehouse.gold.client_features") \
            .toPandas()

        mlflow.log_param("training_rows", len(df_train))

        # Préparation des features
        X = df_train[feature_columns]
        y = df_train["churned"]

        # Entraînement
        model = XGBClassifier(**hyperparams)
        model.fit(X, y)

        # Évaluation
        predictions = model.predict(X_test)
        accuracy = accuracy_score(y_test, predictions)
        mlflow.log_metric("accuracy", accuracy)

        # Sauvegarde du modèle
        mlflow.xgboost.log_model(model, "model")

        return model

# Récupération du snapshot courant pour reproductibilité
table = catalog.load_table("gold.client_features")
current_snapshot = table.current_snapshot().snapshot_id

# Entraînement
model = train_model(current_snapshot)
```

Automatisation CI/CD pour features :

```
# .github/workflows/feature-pipeline.yml
name: Feature Pipeline

on:
  schedule:
    - cron: '0 6 * * *'  # Quotidien à 6h
  workflow_dispatch:

jobs:
  compute-features:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Configure AWS
```

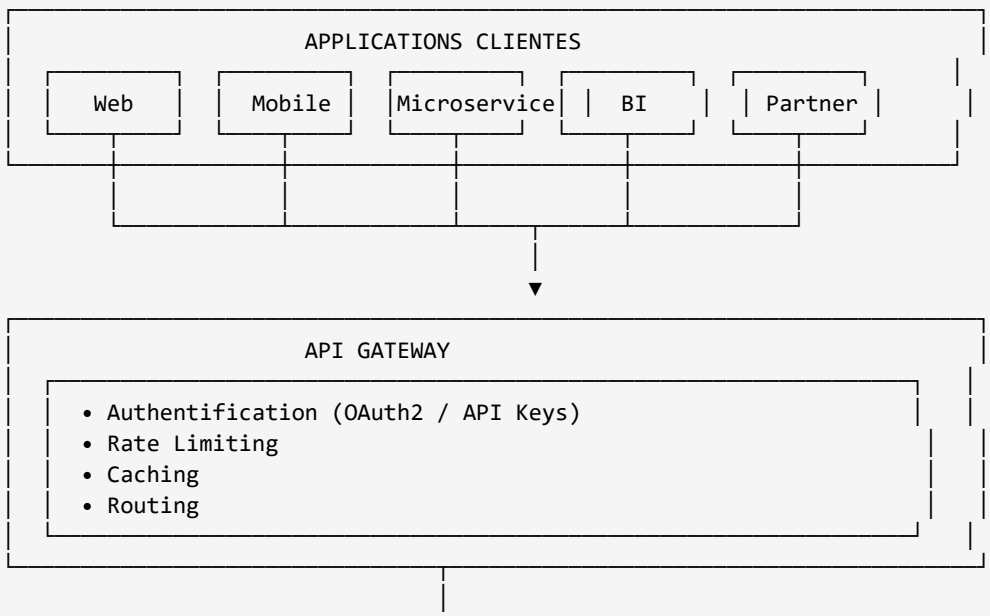
```
        uses: aws-actions/configure-aws-credentials@v2
        with:
          role-to-assume: ${{ secrets.AWS_ROLE_ARN }}
          aws-region: ca-central-1

      - name: Run Feature Computation
        run: |
          spark-submit \
            --master k8s://https://k8s-api.lakehouse.entreprise.ca \
            --conf spark.kubernetes.container.image=features:${{ github.sha }} \
            compute_features.py

      - name: Validate Features
        run: |
          python validate_features.py \
            --table lakehouse.features.client_profil \
            --checks null_check,range_check,freshness_check

      - name: Update Feature Store Registry
        run: |
          feast apply
          feast materialize-incremental $(date +%Y-%m-%dT%H:%M:%S)
```

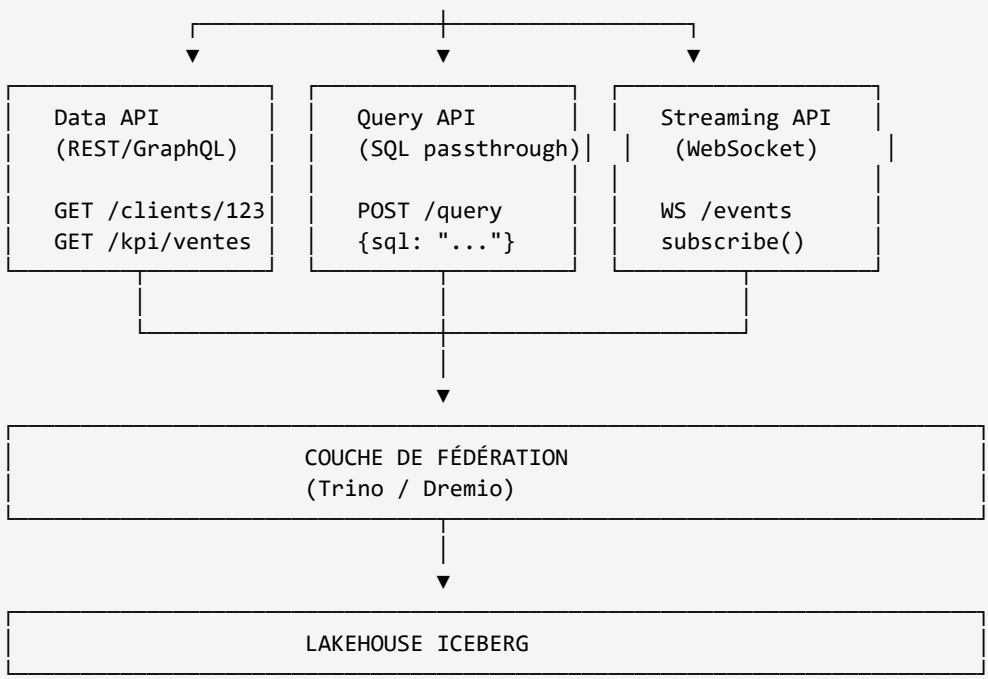
Consommation Applicative

APIs de Données

Les applications modernes nécessitent un accès programmatique aux données du Lakehouse via des APIs bien définies. Plusieurs patterns architecturaux répondent à ces besoins.

Architecture API de données :





API REST avec FastAPI :

```python
from fastapi import FastAPI, Depends, HTTPException, Query
from fastapi.security import OAuth2PasswordBearer
from pydantic import BaseModel
from typing import List, Optional
import trino

app = FastAPI(title="Lakehouse Data API", version="1.0.0")
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

# Pool de connexions Trino
def get_trino_connection():
    return trino.dbapi.connect(
        host="trino.lakehouse.entreprise.ca",
        port=443,
        user="api_service",
        catalog="lakehouse",
        schema="gold",
        http_scheme="https",
        auth=trino.auth.OAuth2Authentication()
    )

# Modèles Pydantic
class ClientSummary(BaseModel):
    client_id: int
    nom: str
    segment: str
    total_achats: float
    derniere_transaction: str

class KPIVentes(BaseModel):
    date: str
    chiffre_affaires: float
    nombre_transactions: int
```

```
    panier_moyen: float

# Endpoints
@app.get("/clients/{client_id}", response_model=ClientSummary)
async def get_client(client_id: int, token: str = Depends(oauth2_scheme)):
    """Récupère le résumé d'un client spécifique."""
    conn = get_trino_connection()
    cursor = conn.cursor()

    cursor.execute("""
        SELECT client_id, nom, segment, total_achats,
               CAST(derniere_transaction AS VARCHAR) as derniere_transaction
        FROM lakehouse.gold.vue_client_360
        WHERE client_id = ?
    """, (client_id,))

    row = cursor.fetchone()
    if not row:
        raise HTTPException(status_code=404, detail="Client non trouvé")

    return ClientSummary(
        client_id=row[0],
        nom=row[1],
        segment=row[2],
        total_achats=row[3],
        derniere_transaction=row[4]
    )

@app.get("/kpi/ventes", response_model=List[KPIVentes])
async def get_kpi_ventes(
    date_debut: str = Query(..., description="Date début (YYYY-MM-DD)"),
    date_fin: str = Query(..., description="Date fin (YYYY-MM-DD)"),
    region: Optional[str] = Query(None, description="Filtrer par région"),
    token: str = Depends(oauth2_scheme)
):
    """Récupère les KPI de ventes pour une période."""
    conn = get_trino_connection()
    cursor = conn.cursor()

    query = """
        SELECT
            CAST(date_transaction AS VARCHAR) as date,
            SUM(montant) as chiffre_affaires,
            COUNT(*) as nombre_transactions,
            AVG(montant) as panier_moyen
        FROM lakehouse.gold.transactions
        WHERE date_transaction BETWEEN ? AND ?
    """
    params = [date_debut, date_fin]

    if region:
        query += " AND region = ?"
        params.append(region)

    query += " GROUP BY date_transaction ORDER BY date_transaction"

    cursor.execute(query, params)

    return [
```

```
        KPIVentes(
            date=row[0],
            chiffre_affaires=row[1],
            nombre_transactions=row[2],
            panier_moyen=row[3]
        )
        for row in cursor.fetchall()
    ]
```

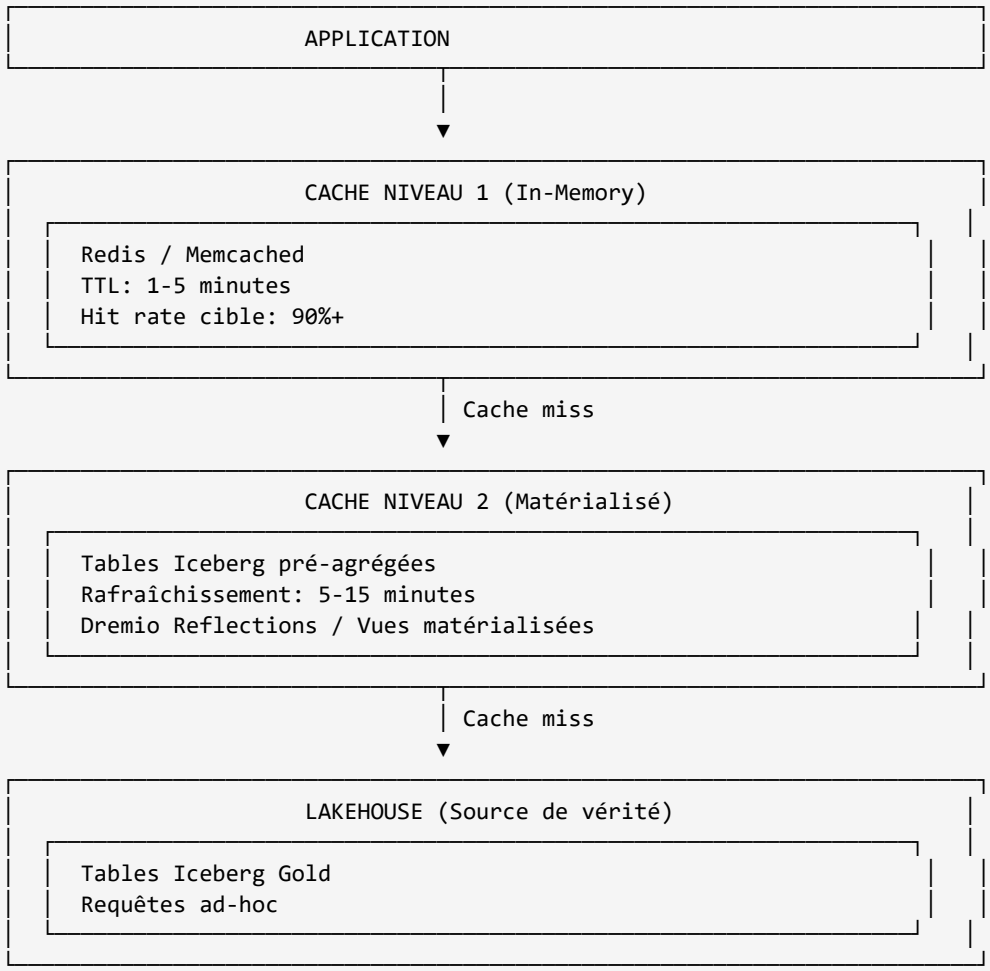
API GraphQL pour requêtes flexibles :

```
import strawberry
from strawberry.fastapi import GraphQLRouter
from typing import List, Optional

@strawberry.type
class Transaction:
    transaction_id: int
    client_id: int
    montant: float
    date_transaction: str
    produit: str

@strawberry.type
class ClientStats:
    client_id: int
    nom: str
    total_achats: float
    nombre_transactions: int
    transactions: List[Transaction]

@strawberry.type
class Query:
    @strawberry.field
    async def client(self, client_id: int) -> Optional[ClientStats]:
        # Implémentation avec requête Trino
        pass

    @strawberry.field
    async def transactions(
        self,
        date_debut: str,
        date_fin: str,
        client_id: Optional[int] = None,
        limit: int = 100
    ) -> List[Transaction]:
        # Implémentation avec requête Trino
        pass

schema = strawberry.Schema(query=Query)
graphql_app = GraphQLRouter(schema)
app.include_router(graphql_app, prefix="/graphql")
```

Accès Temps Réel et Faible Latence

Certaines applications requièrent des latences sub-seconde incompatibles avec les requêtes directes au Lakehouse. Des patterns de cache et de pré-calcul répondent à ces exigences.

Architecture cache multi-niveaux :



Implémentation avec Redis :

```
import redis
import json
from functools import wraps
import hashlib

redis_client = redis.Redis(
    host='redis-cache.lakehouse.entreprise.ca',
    port=6379,
    db=0,
    decode_responses=True
)

def cache_query(ttl_seconds: int = 300):
    """Décorateur pour mise en cache des requêtes."""
    def decorator(func):
        @wraps(func)
        async def wrapper(*args, **kwargs):
            # Génération de la clé de cache
            cache_key = f"query:{func.__name__}:
{hashlib.md5(str(kwargs).encode()).hexdigest()}"
```

```
            # Vérification du cache
            cached = redis_client.get(cache_key)
            if cached:
                return json.loads(cached)

            # Exécution de la requête
            result = await func(*args, **kwargs)

            # Mise en cache
            redis_client.setex(cache_key, ttl_seconds, json.dumps(result))

            return result
        return wrapper
    return decorator

@cache_query(ttl_seconds=60)
async def get_kpi_temps_reel(region: str = None):
    """KPI avec cache de 60 secondes."""
    conn = get_trino_connection()
    cursor = conn.cursor()

    query = """
        SELECT
            SUM(montant) as ca_jour,
            COUNT(*) as nb_transactions,
            COUNT(DISTINCT client_id) as clients_uniques
        FROM lakehouse.gold.transactions
        WHERE date_transaction = CURRENT_DATE
    """
    if region:
        query += f" AND region = '{region}'"

    cursor.execute(query)
    row = cursor.fetchone()

    return {
        "ca_jour": float(row[0] or 0),
        "nb_transactions": row[1],
        "clients_uniques": row[2]
    }
```

Tables pré-agrégées pour performances :

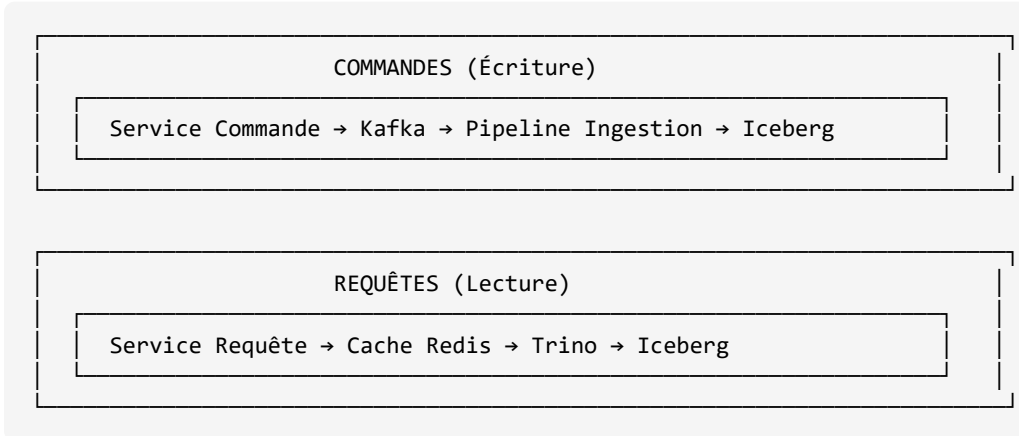
```
-- Table de KPI temps réel rafraîchie toutes les 5 minutes
CREATE TABLE lakehouse.cache.kpi_temps_reel (
    timestamp_calcul TIMESTAMP,
    region VARCHAR,
    ca_jour DECIMAL(15,2),
    nb_transactions BIGINT,
    clients_uniques BIGINT,
    ca_heure_precedente DECIMAL(15,2),
    tendance VARCHAR
)
USING iceberg
PARTITIONED BY (days(timestamp_calcul));

-- Procédure de rafraîchissement
MERGE INTO lakehouse.cache.kpi_temps_reel target
```

```
USING (
    SELECT
        CURRENT_TIMESTAMP as timestamp_calcul,
        region,
        SUM(montant) as ca_jour,
        COUNT(*) as nb_transactions,
        COUNT(DISTINCT client_id) as clients_uniques,
        SUM(CASE WHEN date_transaction >= CURRENT_TIMESTAMP - INTERVAL '1' HOUR
                 THEN montant ELSE 0 END) as ca_heure_precedente,
        CASE
            WHEN SUM(montant) > LAG(SUM(montant)) OVER (PARTITION BY region ORDER BY 1)
            THEN 'hausse'
            ELSE 'baisse'
        END as tendance
    FROM lakehouse.gold.transactions
    WHERE date_transaction = CURRENT_DATE
    GROUP BY region
) source
ON target.region = source.region
   AND DATE(target.timestamp_calcul) = CURRENT_DATE
WHEN MATCHED THEN UPDATE SET *
WHEN NOT MATCHED THEN INSERT *;
```

Microservices et Architecture Événementielle

L'intégration du Lakehouse dans une architecture de microservices requiert des patterns spécifiques pour maintenir la cohérence et la performance.

Pattern CQRS avec Lakehouse :



Service de données événementiel :

```
from kafka import KafkaConsumer, KafkaProducer
import json

class DataChangeNotifier:
    """Notifie les services des changements de données."""

    def __init__(self):
        self.producer = KafkaProducer(
            bootstrap_servers=['kafka:9092'],
```

```
            value_serializer=lambda v: json.dumps(v).encode('utf-8')
        )

    def notify_table_update(self, table: str, snapshot_id: int, rows_affected: int):
        """Publie un événement de mise à jour de table."""
        event = {
            "event_type": "TABLE_UPDATED",
            "table": table,
            "snapshot_id": snapshot_id,
            "rows_affected": rows_affected,
            "timestamp": datetime.utcnow().isoformat()
        }
        self.producer.send('lakehouse.events', value=event)

class DataChangeListener:
    """Écoute les changements de données pour invalidation de cache."""

    def __init__(self):
        self.consumer = KafkaConsumer(
            'lakehouse.events',
            bootstrap_servers=['kafka:9092'],
            value_deserializer=lambda m: json.loads(m.decode('utf-8'))
        )
        self.redis = redis.Redis(host='redis-cache', port=6379)

    def listen(self):
        for message in self.consumer:
            event = message.value
            if event["event_type"] == "TABLE_UPDATED":
                self.invalidate_cache(event["table"])

    def invalidate_cache(self, table: str):
        """Invalide le cache pour une table mise à jour."""
        pattern = f"query:*:{table}:*"
        keys = self.redis.keys(pattern)
        if keys:
            self.redis.delete(*keys)
```

Patterns d'Accès Avancés Iceberg

Time Travel pour Analyse Historique

Le Time Travel Iceberg permet d'interroger les données à n'importe quel point dans le temps, offrant des capacités analytiques puissantes pour l'audit, la conformité et l'analyse de tendances.

Cas d'usage du Time Travel :

Cas d'usage	Requête	Bénéfice
Audit réglementaire	État à date de clôture	Conformité
Debug de pipeline	État avant transformation	Diagnostic
Analyse YoY	Comparaison même date an passé	Tendances
Reproductibilité ML	Snapshot d'entraînement	Science
Rollback analytique	Retour état précédent	Correction

Requêtes Time Travel :

```
-- Requête à un timestamp spécifique
SELECT * FROM lakehouse.gold.transactions
FOR TIMESTAMP AS OF TIMESTAMP '2024-01-01 00:00:00';

-- Requête à un snapshot spécifique
SELECT * FROM lakehouse.gold.transactions
FOR VERSION AS OF 1234567890123456789;

-- Comparaison année sur année
WITH
ventes_2024 AS (
    SELECT
        MONTH(date_transaction) as mois,
        SUM(montant) as ca
    FROM lakehouse.gold.transactions
    FOR TIMESTAMP AS OF TIMESTAMP '2024-12-31 23:59:59'
    WHERE YEAR(date_transaction) = 2024
    GROUP BY MONTH(date_transaction)
),
ventes_2023 AS (
    SELECT
        MONTH(date_transaction) as mois,
        SUM(montant) as ca
    FROM lakehouse.gold.transactions
    FOR TIMESTAMP AS OF TIMESTAMP '2023-12-31 23:59:59'
    WHERE YEAR(date_transaction) = 2023
    GROUP BY MONTH(date_transaction)
)
SELECT
    v24.mois,
    v24.ca as ca_2024,
    v23.ca as ca_2023,
    (v24.ca - v23.ca) / v23.ca * 100 as croissance_pct
FROM ventes_2024 v24
JOIN ventes_2023 v23 ON v24.mois = v23.mois
ORDER BY v24.mois;
```

API Time Travel pour applications :

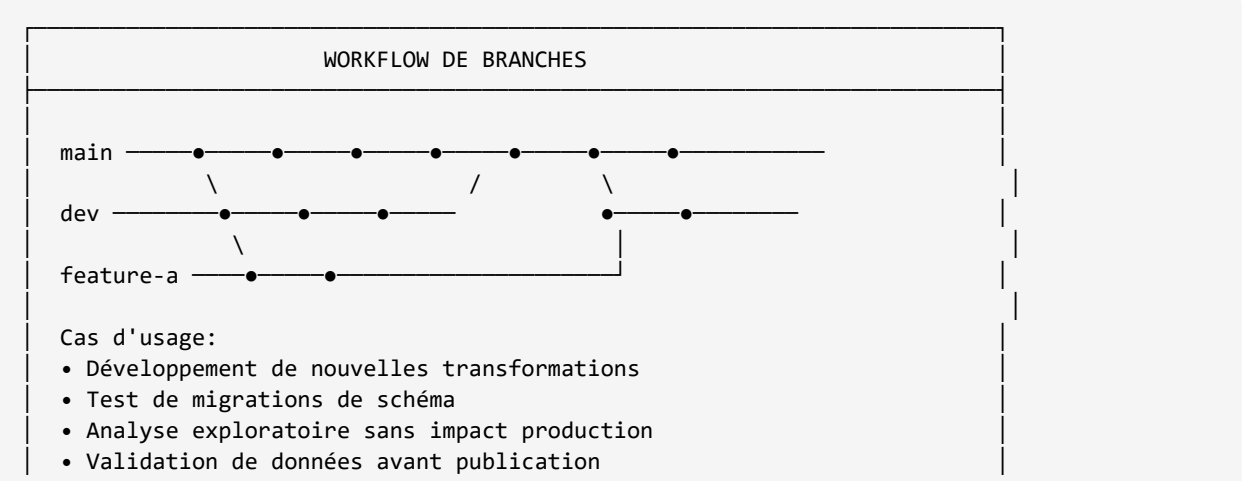
```
class TimeTravel:
    """Service d'accès aux données historiques."""

    def __init__(self, catalog):
```

```
        self.catalog = catalog

    def query_at_timestamp(self, table: str, timestamp: str, sql: str) -> pd.DataFrame:
        """Exécute une requête à un timestamp donné."""
        table_ref = f"{table} FOR TIMESTAMP AS OF TIMESTAMP '{timestamp}'"
        full_sql = sql.replace(f"FROM {table}", f"FROM {table_ref}")
        return self._execute(full_sql)

    def query_at_snapshot(self, table: str, snapshot_id: int, sql: str) -> pd.DataFrame:
        """Exécute une requête à un snapshot donné."""
        table_ref = f"{table} FOR VERSION AS OF {snapshot_id}"
        full_sql = sql.replace(f"FROM {table}", f"FROM {table_ref}")
        return self._execute(full_sql)

    def get_snapshots(self, table: str) -> List[dict]:
        """Liste les snapshots disponibles pour une table."""
        iceberg_table = self.catalog.load_table(table)
        return [
            {
                "snapshot_id": s.snapshot_id,
                "timestamp": s.timestamp_ms,
                "operation": s.operation,
                "summary": s.summary
            }
            for s in iceberg_table.history()
        ]

    def diff_between_snapshots(self, table: str,
                               snapshot_from: int,
                               snapshot_to: int) -> pd.DataFrame:
        """Calcule les différences entre deux snapshots."""
        # Utilise les incremental reads Iceberg
        pass
```

Branches pour Environnements Isolés

Avec des catalogues comme Nessie, les branches permettent de créer des environnements de données isolés sans duplication physique.

Cas d'usage des branches :



```
│                                                                    │
└────────────────────────────────────────────────────────────────────┘
```

Utilisation des branches pour analyse :

```python
# Configuration pour branche de développement
spark_dev = SparkSession.builder \
    .config("spark.sql.catalog.lakehouse.ref", "dev-experiment") \
    .getOrCreate()

# Les requêtes utilisent l'état de la branche dev
df_dev = spark_dev.sql("""
    SELECT * FROM lakehouse.gold.transactions
    WHERE date_transaction = CURRENT_DATE
""")

# Modifications expérimentales (n'affectent pas main)
spark_dev.sql("""
    ALTER TABLE lakehouse.gold.transactions
    ADD COLUMN nouvelle_metrique DECIMAL(10,2)
""")

# Après validation, merge vers main
nessie_client.merge(
    from_ref="dev-experiment",
    to_ref="main",
    message="Ajout métrique validée après tests"
)
```

Lectures Incrémentielles

Les lectures incrémentielles permettent de ne lire que les données modifiées depuis une position donnée, optimisant les pipelines de synchronisation et les flux temps réel.

Pattern de lecture incrémentielle :

```python
def incremental_read(table: str,
                     start_snapshot: int,
                     end_snapshot: int = None) -> DataFrame:
    """Lit uniquement les modifications entre deux snapshots."""

    reader = spark.read.format("iceberg")

    if end_snapshot:
        # Modifications entre deux snapshots
        reader = reader \
            .option("start-snapshot-id", start_snapshot) \
            .option("end-snapshot-id", end_snapshot)
    else:
        # Modifications depuis un snapshot jusqu'à maintenant
        reader = reader \
            .option("start-snapshot-id", start_snapshot)

    return reader.load(f"lakehouse.{table}")
```

```python
# Exemple: synchronisation incrémentielle vers système externe
class IncrementalSync:
    def __init__(self, table: str):
        self.table = table
        self.state_store = redis.Redis()

    def sync(self):
        # Récupération du dernier snapshot synchronisé
        last_snapshot = self.state_store.get(f"sync:{self.table}:last_snapshot")
        last_snapshot = int(last_snapshot) if last_snapshot else None

        # Récupération du snapshot courant
        iceberg_table = catalog.load_table(self.table)
        current_snapshot = iceberg_table.current_snapshot().snapshot_id

        if last_snapshot == current_snapshot:
            print("Aucune modification")
            return

        # Lecture incrémentielle
        if last_snapshot:
            df_changes = incremental_read(self.table, last_snapshot, current_snapshot)
        else:
            # Premier sync: lecture complète
            df_changes = spark.read.table(f"lakehouse.{self.table}")

        # Synchronisation vers système cible
        self.sync_to_target(df_changes)

        # Mise à jour de l'état
        self.state_store.set(f"sync:{self.table}:last_snapshot", current_snapshot)
```

Streaming incrémental avec Spark :

```python
# Lecture streaming des modifications Iceberg
df_stream = spark.readStream \
    .format("iceberg") \
    .option("stream-from-timestamp", "2024-01-01T00:00:00Z") \
    .load("lakehouse.gold.transactions")

# Traitement des modifications en temps réel
query = df_stream \
    .writeStream \
    .foreachBatch(process_incremental_batch) \
    .option("checkpointLocation", "s3://checkpoints/incremental-sync") \
    .start()
```

Gouvernance de la Consommation

Quotas et Limites de Ressources

La gestion des ressources prévient les abus et garantit une expérience équitable pour tous les consommateurs.

Configuration des quotas Trino :

```
# etc/resource-groups.json
{
  "rootGroups": [
    {
      "name": "global",
      "softMemoryLimit": "80%",
      "hardConcurrencyLimit": 100,
      "maxQueued": 1000,
      "subGroups": [
        {
          "name": "adhoc",
          "softMemoryLimit": "30%",
          "hardConcurrencyLimit": 30,
          "maxQueued": 200,
          "schedulingPolicy": "fair",
          "schedulingWeight": 1
        },
        {
          "name": "pipeline",
          "softMemoryLimit": "40%",
          "hardConcurrencyLimit": 20,
          "maxQueued": 100,
          "schedulingPolicy": "fair",
          "schedulingWeight": 2
        },
        {
          "name": "bi_reports",
          "softMemoryLimit": "20%",
          "hardConcurrencyLimit": 40,
          "maxQueued": 500,
          "schedulingPolicy": "fair",
          "schedulingWeight": 1
        },
        {
          "name": "api_service",
          "softMemoryLimit": "10%",
          "hardConcurrencyLimit": 50,
          "maxQueued": 1000,
          "schedulingPolicy": "fair",
          "schedulingWeight": 3
        }
      ]
    }
  ],
  "selectors": [
    {
      "group": "global.pipeline",
      "user": "pipeline_.*"
```

```
    },
    {
      "group": "global.bi_reports",
      "source": "tableau|powerbi"
    },
    {
      "group": "global.api_service",
      "user": "api_service"
    },
    {
      "group": "global.adhoc"
    }
  ]
}
```

Limites par utilisateur :

```
-- Configuration des limites utilisateur
CREATE RESOURCE GROUP user_standard WITH (
    cpu_quota_period = '1m',
    cpu_quota = '500ms',
    memory_limit = '4GB',
    query_max_memory = '2GB',
    query_max_execution_time = '10m',
    query_max_cpu_time = '5m'
);

-- Attribution à un groupe d'utilisateurs
GRANT RESOURCE GROUP user_standard TO ROLE data_analyst;
```

Contrôle d'Accès Granulaire

Le contrôle d'accès doit refléter les besoins métier tout en minimisant la surface d'exposition des données sensibles.

Matrice d'accès par profil :

Profil	Bronze	Silver	Gold	Features	Cache
Dirigeant	✗	✗	Lecture (agrégé)	✗	Lecture
Analyste affaires	✗	✗	Lecture	✗	Lecture
Analyste données	✗	Lecture	Lecture	✗	Lecture
Data Scientist	Lecture	Lecture	Lecture	Lecture/Écriture	Lecture
Data Engineer	Lecture/Écriture	Lecture/Écriture	Lecture/Écriture	Lecture/Écriture	Admin
Application	✗	✗	Lecture (spécifique)	Lecture	Lecture

Implémentation des politiques :

```
-- Politique pour analystes d'affaires
CREATE ROLE business_analyst;
GRANT USAGE ON CATALOG lakehouse TO ROLE business_analyst;
GRANT USAGE ON SCHEMA lakehouse.gold TO ROLE business_analyst;
GRANT USAGE ON SCHEMA lakehouse.semantic TO ROLE business_analyst;
GRANT SELECT ON ALL TABLES IN SCHEMA lakehouse.gold TO ROLE business_analyst;
GRANT SELECT ON ALL TABLES IN SCHEMA lakehouse.semantic TO ROLE business_analyst;

-- Masquage automatique pour analystes
ALTER TABLE lakehouse.gold.clients
ALTER COLUMN courriel SET MASK
    CASE WHEN current_role() = 'data_admin' THEN courriel
         ELSE regexp_replace(courriel, '(.{2}).*@', '$1***@') END;

-- Filtrage ligne pour accès régional
CREATE ROW ACCESS POLICY regional_access ON lakehouse.gold.transactions
FOR SELECT TO ROLE regional_analyst
USING region IN (
    SELECT region_autorisee
    FROM lakehouse.security.autorisations_regionales
    WHERE utilisateur = current_user()
);
```

Monitoring et Observabilité

La surveillance de la consommation permet d'identifier les problèmes de performance, les abus et les opportunités d'optimisation.

Métriques de consommation :

```
# prometheus-rules.yaml
groups:
  - name: consumption-metrics
    rules:
      # Requêtes par profil utilisateur
      - record: trino_queries_by_user_group
        expr: sum(rate(trino_query_completed_total[5m])) by (user_group)

      # Latence par type de requête
      - record: trino_query_latency_p99
        expr: histogram_quantile(0.99, rate(trino_query_execution_time_bucket[5m]))

      # Volume de données scannées
      - record: trino_data_scanned_bytes_total
        expr: sum(rate(trino_physical_input_bytes_total[5m]))

      # Alertes
      - alert: HighQueryLatency
        expr: trino_query_latency_p99 > 30
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "Latence de requête élevée"
```

```
      - alert: ExcessiveDataScan
        expr: trino_data_scanned_bytes_total > 1e12
        for: 10m
        labels:
          severity: warning
        annotations:
          summary: "Scan de données excessif détecté"
```

Dashboard de consommation :

```
-- Vue pour monitoring de la consommation
CREATE VIEW lakehouse.monitoring.consumption_stats AS
SELECT
    DATE_TRUNC('hour', query_start_time) as heure,
    user_group,
    COUNT(*) as nb_requetes,
    SUM(CASE WHEN state = 'FINISHED' THEN 1 ELSE 0 END) as succes,
    SUM(CASE WHEN state = 'FAILED' THEN 1 ELSE 0 END) as echecs,
    AVG(execution_time_ms) as latence_moyenne_ms,
    PERCENTILE_CONT(0.99) WITHIN GROUP (ORDER BY execution_time_ms) as latence_p99_ms,
    SUM(physical_input_bytes) / 1e9 as donnees_scannees_gb,
    SUM(output_rows) as lignes_retournees
FROM system.runtime.queries
WHERE query_start_time >= CURRENT_TIMESTAMP - INTERVAL '7' DAY
GROUP BY DATE_TRUNC('hour', query_start_time), user_group;
```

Études de Cas Canadiennes

Secteur Manufacturier : Équipementier Automobile

Étude de cas : Équipementier automobile ontarien

Secteur : Fabrication automobile

Défi : Démocratiser l'accès aux données de production pour 500+ ingénieurs et gestionnaires répartis sur 12 usines au Canada et aux États-Unis. Les données de production, qualité et maintenance étaient silotées dans des systèmes MES (Manufacturing Execution System) propriétaires.

Solution : Déploiement d'un Lakehouse Iceberg consolidant les données de tous les sites avec une couche sémantique Dremio exposant des vues métier standardisées. Tableaux de bord Power BI pour les gestionnaires, accès notebook pour les ingénieurs qualité, API pour les systèmes d'alerte.

Architecture de consommation :

```
┌─────────────────────────────────────────────────────────────┐
│                    CONSOMMATEURS                            │
│  Gestionnaires (150) │ Ingénieurs (300) │ Systèmes (50)     │
│  Power BI            │ Jupyter + SQL    │ API REST          │
```

```
└──────────────────────────────────────────────────────────────┘
                              │
                              ▼
┌──────────────────────────────────────────────────────────────┐
│                   COUCHE SÉMANTIQUE (Dremio)                 │
│  • Vues standardisées par usine                              │
│  • KPI qualité temps réel                                    │
│  • Historique maintenance                                    │
│  • Reflections pour tableaux de bord                         │
└──────────────────────────────────────────────────────────────┘
                              │
                              ▼
┌──────────────────────────────────────────────────────────────┐
│                   LAKEHOUSE ICEBERG (S3)                     │
│  12 usines × 200 tables = 2400 tables                        │
│  5 ans d'historique = 50 To                                  │
└──────────────────────────────────────────────────────────────┘
```

Résultats :

- Temps d'accès aux données réduit de 2 jours à 5 minutes
- 85% des rapports désormais en self-service
- Détection d'anomalies qualité accélérée de 70%
- Économies de 2M$/an en coûts d'analyse externalisée

Secteur Assurance : Mutuelle Pancanadienne

Étude de cas : Compagnie d'assurance mutuelle

Secteur : Assurance IARD (Incendie, Accidents, Risques Divers)

Défi : Permettre aux actuaires et analystes de risques d'accéder aux données de sinistres historiques (20 ans) pour modélisation, tout en respectant les exigences de confidentialité des assurés. Les scientifiques de données avaient besoin d'accès aux données brutes pour l'entraînement de modèles de détection de fraude.

Solution : Lakehouse Iceberg avec contrôle d'accès granulaire via Apache Ranger. Données personnelles masquées automatiquement selon le rôle. Feature Store Iceberg pour les modèles ML avec versionnement des features.

Contrôle d'accès :

```sql
-- Actuaires: accès agrégé uniquement
SELECT
    region,
    type_sinistre,
    annee,
    SUM(montant_indemnise) as total_indemnites,
    COUNT(*) as nb_sinistres,
    AVG(montant_indemnise) as indemnite_moyenne
FROM lakehouse.gold.sinistres_anonymises
GROUP BY region, type_sinistre, annee;
```

```
-- Data Scientists: accès détail avec masquage PII
SELECT
    sinistre_id,
    hash(assure_id) as assure_hash,  -- Pseudonymisé
    '***' as nom_assure,              -- Masqué
    type_sinistre,
    montant_reclame,
    montant_indemnise,
    features_fraude
FROM lakehouse.silver.sinistres_ml;
```

Résultats :

- 200 actuaires avec accès self-service sécurisé
- Modèle de fraude réduisant les pertes de 15%
- Conformité aux exigences du BSIF et de l'AMF
- Temps de développement des modèles réduit de 60%

Secteur Public : Agence Provinciale de Transport

Étude de cas : Société de transport provincial

Secteur : Transport public

Défi : Analyser les données de 50 millions de trajets annuels pour optimiser les horaires et capacités. Exposition de données ouvertes au public tout en protégeant les données de validation des titres de transport individuels.

Solution : Architecture de consommation à trois niveaux : données ouvertes (agrégées) via API publique, données opérationnelles via Power BI pour gestionnaires, données détaillées via notebooks pour équipe analytique interne.

Architecture :

```
┌─────────────────────────────────────────────────────────┐
│  PUBLIC (Données ouvertes)                              │
│  • API REST publique                                    │
│  • Fréquentation par station/heure (agrégé)             │
│  • Pas de données individuelles                         │
└─────────────────────────────────────────────────────────┘
                              │
┌─────────────────────────────────────────────────────────┐
│  OPÉRATIONNEL (Gestionnaires)                           │
│  • Power BI avec Row-Level Security                     │
│  • Données par ligne/station gérée                      │
│  • Rafraîchissement horaire                             │
└─────────────────────────────────────────────────────────┘
                              │
┌─────────────────────────────────────────────────────────┐
│  ANALYTIQUE (Équipe interne)                            │
```

```
│  • Jupyter + Spark                                          │
│  • Données détaillées pseudonymisées                        │
│  • Modélisation prédictive                                  │
└─────────────────────────────────────────────────────────────┘
```

Résultats :

- 10 000+ téléchargements/mois des données ouvertes
- Optimisation des horaires réduisant les temps d'attente de 12%
- Conformité aux lois d'accès à l'information
- Économies de 5M$/an en optimisation des ressources

Secteur Détail : Épicier Québécois

Étude de cas : Chaîne d'épiceries québécoise

Secteur : Commerce alimentaire de détail

Défi : Unifier les données de 400 magasins, du commerce en ligne et du programme de fidélité pour personnalisation des offres. Les gestionnaires de catégories avaient besoin d'analyses en temps réel pendant les négociations fournisseurs.

Solution : Lakehouse Iceberg avec couche de consommation multi-niveaux. Dremio avec Reflections pour accélération des tableaux de bord. Feature Store pour le moteur de recommandation personnalisée.

Métriques de consommation :

```
Consommateurs:
- 50 gestionnaires catégories: Power BI (temps réel)
- 20 analystes merchandising: SQL + Tableau
- 5 data scientists: Notebooks + Feature Store
- 1 système recommandation: API (1000 req/sec)

Performance:
- Latence tableaux de bord: < 2 secondes (avec Reflections)
- Latence API recommandation: < 100ms (avec cache Redis)
- Fraîcheur données: 15 minutes (streaming lakehouse)
```

Résultats :

- Personnalisation augmentant le panier moyen de 8%
- Réduction des ruptures de stock de 25%
- Temps de génération des rapports catégorie de 4h à 30 secondes
- ROI de 300% la première année

Bonnes Pratiques et Recommandations

Architecture de Référence par Profil

Pour analystes d'affaires :

```
Recommandations:
- Couche sémantique avec tables dénormalisées
- Nommage métier explicite
- Documentation intégrée
- Métriques pré-calculées
- Accès Power BI/Tableau optimisé
- Reflections pour performance

Anti-patterns à éviter:
- Exposition directe des tables techniques
- Jointures complexes requises
- Nommage technique (id_fk, amt_ttc_ht)
```

Pour data scientists :

```
Recommandations:
- Accès direct aux tables Silver pour exploration
- Feature Store centralisé
- Time Travel pour reproductibilité
- Branches pour expérimentation
- Intégration MLflow/Weights&Biases

Anti-patterns à éviter:
- Copies de données manuelles
- Features calculées localement
- Manque de versionnement
```

Pour applications :

```
Recommandations:
- Cache multi-niveaux (Redis + matérialisé)
- API avec rate limiting
- Tables pré-agrégées pour requêtes fréquentes
- Circuit breaker pour résilience
- Monitoring latence et disponibilité

Anti-patterns à éviter:
- Requêtes ad-hoc depuis applications
- Absence de cache
- Dépendance directe au Lakehouse pour SLA critique
```

Checklist de Mise en Production

Préparation :

- ☐ Profils de consommateurs identifiés et documentés
- ☐ Exigences de latence définies par profil
- ☐ Volumes et fréquences d'accès estimés

- ☐ Politiques de sécurité définies

Infrastructure :

- ☐ Couche sémantique configurée
- ☐ Cache multi-niveaux déployé
- ☐ API documentée et versionnée
- ☐ Quotas et limites configurés

Sécurité :

- ☐ Contrôle d'accès par rôle implémenté
- ☐ Masquage des données sensibles actif
- ☐ Row-level security si nécessaire
- ☐ Audit de consommation activé

Monitoring :

- ☐ Métriques de latence collectées
- ☐ Alertes configurées
- ☐ Dashboard de consommation disponible
- ☐ Processus d'optimisation établi

Matrice de Décision Outils

Besoin	Solution Recommandée	Alternative
BI entreprise	Power BI + Trino	Tableau + Dremio
Self-service	Dremio (couche sémantique)	Superset
Data Science	PyIceberg + Notebooks	PySpark
Feature Store	Feast + Iceberg	Tecton
API faible latence	FastAPI + Redis + Trino	GraphQL + Cache
Temps réel	Kafka + Flink + Iceberg	Spark Streaming

Conclusion

La couche de consommation constitue l'interface critique entre votre Data Lakehouse Apache Iceberg et les utilisateurs qui en extraient la valeur. Son succès repose sur une compréhension fine des besoins distincts de chaque profil de consommateur — des dirigeants consultant des KPI synthétiques aux applications interrogeant le Lakehouse en temps réel — et sur une architecture adaptée à cette diversité.

Apache Iceberg apporte des capacités différenciantes qui transforment l'expérience de consommation. Le Time Travel permet aux analystes d'explorer l'historique sans intervention technique. L'évolution de schéma garantit la continuité des requêtes malgré les modifications structurelles. Les lectures incrémentielles optimisent les synchronisations. Ces fonctionnalités, souvent invisibles pour les utilisateurs finaux, élèvent considérablement la qualité et la fiabilité de l'accès aux données.

La démocratisation des données — objectif central de nombreuses organisations — ne se résume pas à « donner accès ». Elle requiert une couche sémantique traduisant la complexité technique en concepts métier, des mécanismes de cache garantissant des performances acceptables, et une gouvernance granulaire protégeant les données sensibles sans entraver les usages légitimes. Les études de cas canadiennes présentées démontrent que cet équilibre est atteignable dans des contextes variés.

Les tendances émergentes — feature stores centralisés, APIs de données standardisées, intégration MLOps native — continueront d'enrichir les patterns de consommation. Les organisations investissant dans une couche de consommation bien architecturée se positionnent favorablement pour adopter ces évolutions sans refonte majeure.

Le chapitre suivant aborde la maintenance en production du Lakehouse, où nous examinerons les opérations quotidiennes — compaction, expiration des snapshots, optimisation — garantissant performance et fiabilité dans la durée.

Résumé

Profils de consommateurs :

- **Dirigeants** : KPI agrégés, latence < 1s, interfaces visuelles
- **Analystes affaires** : Tableaux de bord, drill-down, self-service
- **Analystes données** : SQL complexe, jointures, exploration
- **Data Scientists** : Notebooks, DataFrames, reproductibilité
- **Applications** : APIs faible latence, haute disponibilité
- **Pipelines** : Accès programmatique, batch/streaming

Intégration BI :

- Power BI : Import, DirectQuery ou Direct Lake (Fabric)
- Tableau : JDBC avec extraits pour performance
- Self-service : Tables dénormalisées, nommage métier, documentation

Science des données :

- PyIceberg et PySpark pour accès DataFrame
- Time Travel pour reproductibilité des expériences
- Feature Stores avec Iceberg comme backend
- Intégration MLflow pour traçabilité

Consommation applicative :

- APIs REST/GraphQL avec FastAPI
- Cache multi-niveaux (Redis + matérialisé)
- Pattern CQRS pour séparation lecture/écriture
- Rate limiting et circuit breakers

Patterns Iceberg avancés :

- Time Travel pour analyse historique et audit
- Branches (Nessie) pour environnements isolés
- Lectures incrémentielles pour synchronisation optimisée

Gouvernance :

- Quotas et resource groups par profil
- Contrôle d'accès granulaire (table, colonne, ligne)
- Masquage automatique des données sensibles
- Monitoring et alerting de la consommation

Ce chapitre établit les fondations de l'accès aux données de votre Lakehouse. Le chapitre suivant, « Maintenance en Production », détaille les opérations quotidiennes garantissant performance et fiabilité sur le long terme.

Chapitre IV.10 - Maintenir un Lakehouse Iceberg en Production

Introduction

La mise en production d'un Data Lakehouse Apache Iceberg ne constitue pas une destination mais le début d'un voyage opérationnel continu. Les tables Iceberg, contrairement aux systèmes de fichiers statiques, sont des organismes vivants qui accumulent des fichiers, génèrent des métadonnées et évoluent au rythme des écritures et modifications. Sans maintenance proactive, cette croissance organique dégrade progressivement les performances, gonfle les coûts de stockage et compromet la fiabilité des requêtes. La différence entre un Lakehouse performant et un système léthargique réside souvent dans la qualité des pratiques opérationnelles plutôt que dans l'architecture initiale.

Apache Iceberg intègre des mécanismes sophistiqués de gestion du cycle de vie des données — compaction, expiration des snapshots, réécriture des fichiers de manifeste — mais ces fonctionnalités ne s'activent pas automatiquement. Elles requièrent une orchestration délibérée, un paramétrage adapté à votre contexte et une surveillance continue. Les organisations qui excellent dans l'exploitation de leur Lakehouse ont développé des pratiques opérationnelles matures, combinant automatisation intelligente, monitoring proactif et procédures de réponse aux incidents bien rodées.

Ce chapitre vous guide dans l'établissement de pratiques de maintenance robustes pour votre Lakehouse Iceberg en production. Nous examinerons en détail les opérations essentielles — compaction, expiration, optimisation —, leurs paramètres critiques et leurs stratégies d'orchestration. Nous aborderons le monitoring et l'observabilité, la gestion des incidents et les procédures de récupération. Les études de cas canadiennes illustreront comment des organisations de différents secteurs ont structuré leurs opérations pour maintenir performance et fiabilité dans la durée.

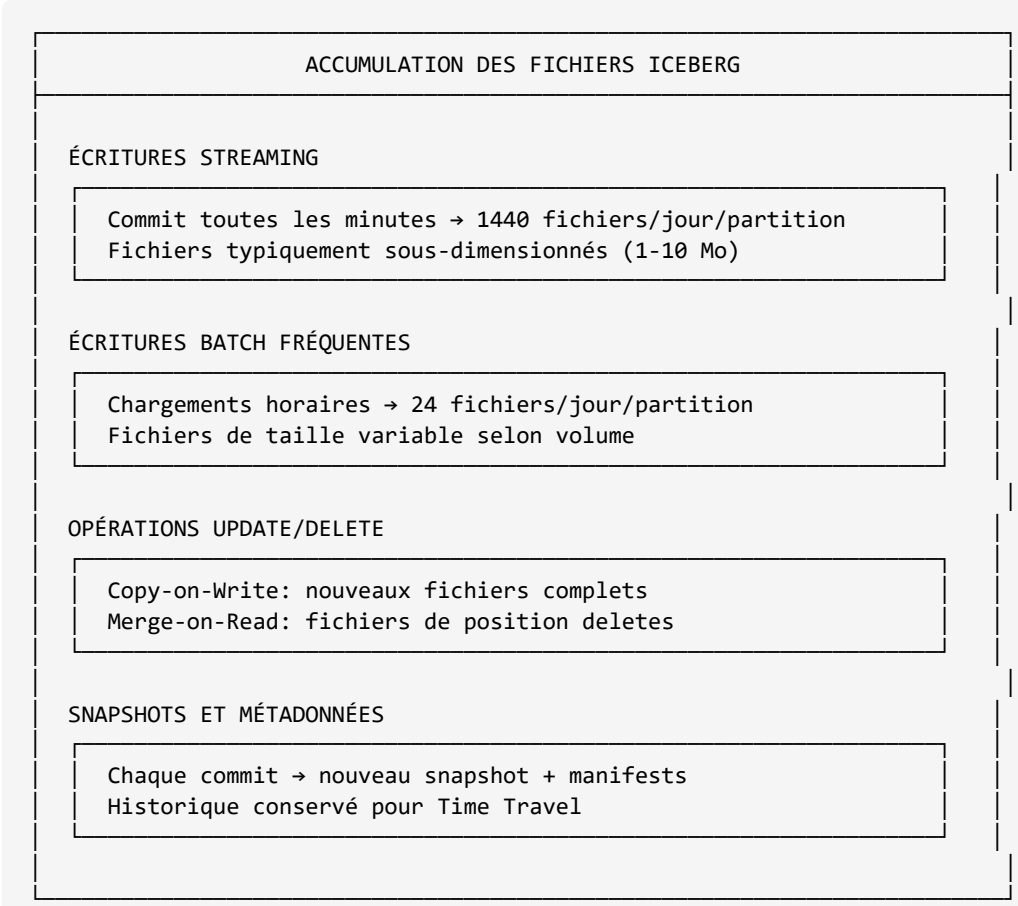
L'enjeu dépasse la simple exécution de tâches techniques. La maintenance d'un Lakehouse en production façonne l'expérience utilisateur quotidienne, influence les coûts d'exploitation et détermine la confiance que l'organisation accorde à sa plateforme de données. Un Lakehouse bien maintenu devient un actif stratégique ; mal maintenu, il devient une source de frustration et de risques.

Anatomie de la Maintenance Iceberg

Comprendre l'Accumulation des Fichiers

Chaque opération d'écriture dans une table Iceberg génère de nouveaux fichiers sans modifier les existants — c'est le principe fondamental de l'immutabilité qui garantit les propriétés ACID. Cette approche, bien que puissante, crée une accumulation naturelle de fichiers qui nécessite une gestion active.

Sources d'accumulation :



Impact de l'accumulation non gérée :

Symptôme	Cause	Impact
Ralentissement des requêtes	Trop de petits fichiers	Overhead de planification
Augmentation des coûts S3	Fichiers orphelins	Stockage inutile
Lenteur du listage de tables	Manifests volumineux	Latence metadata
Échecs de commit	Conflits de concurrence	Fiabilité dégradée
Time Travel défaillant	Snapshots corrompus	Perte de fonctionnalité

Cycle de Vie des Objets Iceberg

Comprendre le cycle de vie des différents objets Iceberg est essentiel pour une maintenance efficace.

Fichiers de données :

- Créés lors des écritures (INSERT, UPDATE via Copy-on-Write)
- Référencés par les manifests
- Deviennent orphelins après compaction ou expiration de snapshot
- Supprimables uniquement après déréférencement complet

Fichiers de manifeste (Manifest Files) :

- Listent les fichiers de données avec leurs statistiques

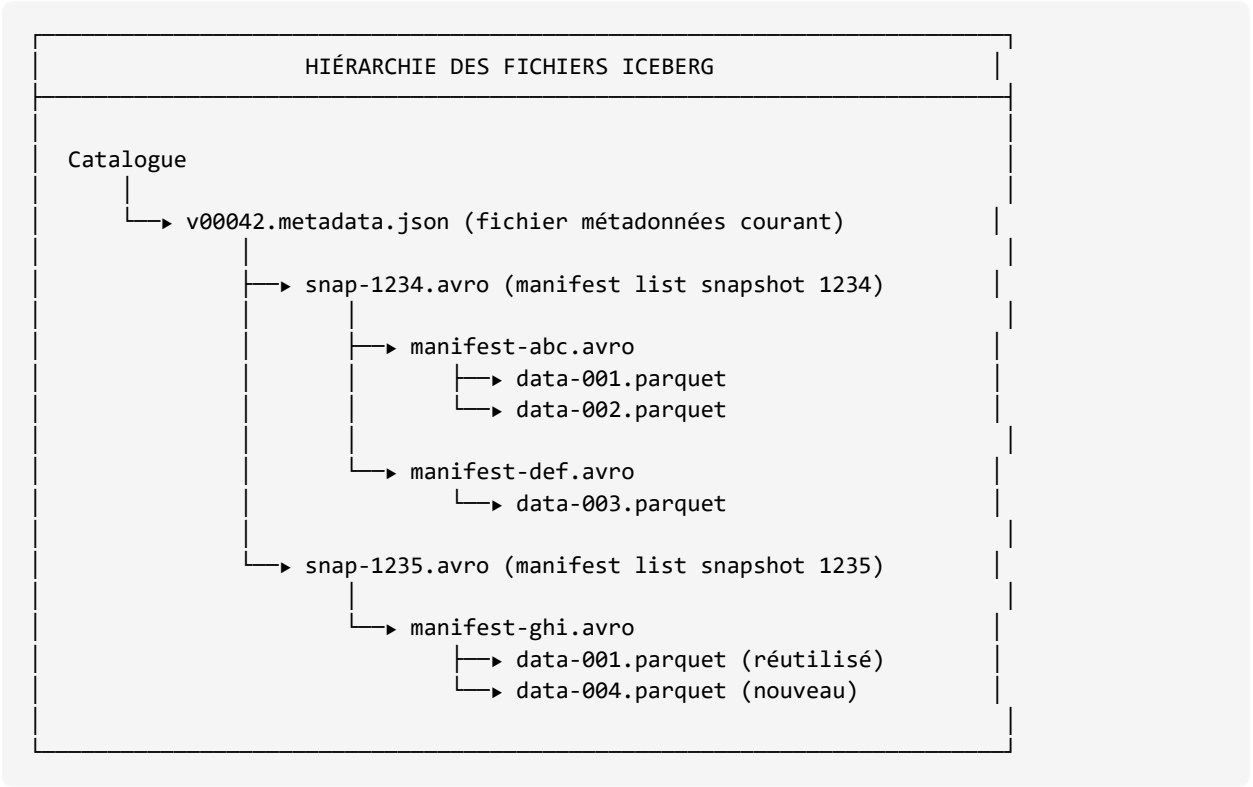
- Créés lors des commits
- Peuvent être réécrits pour optimisation
- Orphelins après réécriture ou expiration

Listes de manifestes (Manifest Lists) :

- Pointent vers les manifests pour un snapshot
- Un fichier par snapshot
- Supprimables avec le snapshot associé

Fichiers de métadonnées (Metadata Files) :

- Décrivent l'état complet de la table
- Un nouveau fichier par commit
- Contiennent l'historique des snapshots
- Le catalogue pointe vers le fichier courant



Opérations de Maintenance Essentielles

Les opérations de maintenance Iceberg se répartissent en plusieurs catégories selon leur fréquence et leur impact.

Opération	Fréquence	Impact Performance	Ressources	Priorité
Compaction	Quotidienne	Élevé (amélioration)	Moyen-Élevé	Critique
Expiration snapshots	Quotidienne	Moyen (stockage)	Faible	Haute
Nettoyage orphelins	Hebdomadaire	Faible (stockage)	Faible	Moyenne
Réécriture manifests	Mensuelle	Moyen (metadata)	Faible	Moyenne
Optimisation tri	Selon besoin	Élevé (requêtes)	Élevé	Variable
Collecte statistiques	Après changements	Moyen (optimiseur)	Moyen	Haute

Compaction des Fichiers de Données

Principes et Stratégies

La compaction constitue l'opération de maintenance la plus critique pour les performances d'un Lakehouse Iceberg. Elle consolide les petits fichiers en fichiers plus volumineux, réduisant l'overhead de planification des requêtes et optimisant les lectures.

Problème des petits fichiers :

```
Avant compaction (streaming toutes les minutes pendant 24h):
├── data-00001.parquet (5 Mo)
├── data-00002.parquet (3 Mo)
├── data-00003.parquet (8 Mo)
├── ... (1437 fichiers)
└── data-01440.parquet (4 Mo)

Total: 1440 fichiers, ~6 Go, taille moyenne 4 Mo

Après compaction:
├── data-compacted-001.parquet (512 Mo)
├── data-compacted-002.parquet (512 Mo)
├── ... (10 fichiers)
└── data-compacted-012.parquet (480 Mo)

Total: 12 fichiers, ~6 Go, taille moyenne 500 Mo
```

Impact sur les requêtes :

```
Requête: SELECT SUM(montant) FROM transactions WHERE date = '2024-01-15'

Avant compaction:
- Planification: lecture de 1440 entrées de manifest
- Exécution: ouverture de 1440 fichiers
- Temps total: 45 secondes

Après compaction:
- Planification: lecture de 12 entrées de manifest
```

```
- Exécution: ouverture de 12 fichiers
- Temps total: 3 secondes

Amélioration: 15x
```

Procédure de Compaction avec Spark

Compaction de base (bin-packing) :

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("IcebergCompaction") \
    .config("spark.sql.extensions",
            "org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions") \
    .config("spark.sql.catalog.lakehouse",
            "org.apache.iceberg.spark.SparkCatalog") \
    .config("spark.sql.catalog.lakehouse.type", "rest") \
    .config("spark.sql.catalog.lakehouse.uri",
            "https://catalog.lakehouse.entreprise.ca") \
    .getOrCreate()

# Compaction simple - regroupe les petits fichiers
spark.sql("""
    CALL lakehouse.system.rewrite_data_files(
        table => 'gold.transactions',
        options => map(
            'target-file-size-bytes', '536870912',
            'min-file-size-bytes', '104857600',
            'max-file-size-bytes', '805306368',
            'min-input-files', '5',
            'max-concurrent-file-group-rewrites', '10',
            'partial-progress.enabled', 'true',
            'partial-progress.max-commits', '10'
        )
    )
""")
```

Paramètres de compaction :

Paramètre	Description	Valeur recommandée
`target-file-size-bytes`	Taille cible des fichiers	512 Mo (536870912)
`min-file-size-bytes`	Seuil minimum pour inclusion	100 Mo
`max-file-size-bytes`	Taille maximale autorisée	768 Mo
`min-input-files`	Nombre min de fichiers pour déclencher	5
`max-concurrent-file-group-rewrites`	Parallélisme	10-20
`partial-progress.enabled`	Commits intermédiaires	true
`partial-progress.max-commits`	Limite commits partiels	10

Compaction avec filtrage de partitions :

```
# Compaction ciblée sur partitions récentes
from datetime import datetime, timedelta

date_limite = (datetime.now() - timedelta(days=7)).strftime('%Y-%m-%d')

spark.sql(f"""
    CALL lakehouse.system.rewrite_data_files(
        table => 'gold.transactions',
        strategy => 'binpack',
        where => 'date_partition >= DATE ''{date_limite}''',
        options => map(
            'target-file-size-bytes', '536870912',
            'min-input-files', '3'
        )
    )
""")
```

Compaction avec Tri (Sort Compaction)

La compaction avec tri réorganise physiquement les données selon un ordre optimal pour les patterns de requêtes, améliorant significativement l'efficacité du filtrage.

Compaction avec tri simple :

```
# Tri par colonnes fréquemment filtrées
spark.sql("""
    CALL lakehouse.system.rewrite_data_files(
        table => 'gold.transactions',
        strategy => 'sort',
        sort_order => 'date_transaction ASC NULLS LAST, client_id ASC NULLS LAST',
        options => map(
            'target-file-size-bytes', '536870912',
            'rewrite-all', 'true'
        )
    )
""")
```

Compaction Z-Order pour requêtes multi-colonnes :

Le Z-Order organise les données pour optimiser les requêtes filtrant sur plusieurs colonnes simultanément — typique des analyses ad-hoc.

```
# Z-Order sur colonnes de filtrage fréquent
spark.sql("""
    CALL lakehouse.system.rewrite_data_files(
        table => 'gold.transactions',
        strategy => 'sort',
        sort_order => 'zorder(region, categorie_produit, client_id)',
        options => map(
            'target-file-size-bytes', '536870912',
            'rewrite-all', 'true'
        )
    )
""")
```

> **Performance**
>
> Le Z-Order peut améliorer les performances de requêtes filtrant sur 2-3 colonnes de 3× à 10×. Cependant, il dégrade légèrement les requêtes filtrant sur une seule colonne par rapport à un tri simple. Analysez vos patterns de requêtes avant de choisir entre tri simple et Z-Order. Pour les tables avec un pattern de filtrage dominant (ex: toujours par date), le tri simple reste préférable.

Stratégies de Compaction par Type de Table

Tables de streaming (haute fréquence d'écriture) :

```
# Configuration agressive pour streaming
compaction_config_streaming = {
    'target-file-size-bytes': '268435456',  # 256 Mo (plus petit pour fraîcheur)
    'min-input-files': '3',  # Seuil bas pour compaction fréquente
    'max-concurrent-file-group-rewrites': '20',
    'partial-progress.enabled': 'true',
    'partial-progress.max-commits': '5'
}

# Exécution toutes les heures
spark.sql(f"""
    CALL lakehouse.system.rewrite_data_files(
        table => 'bronze.events_streaming',
        strategy => 'binpack',
        where => 'event_hour >= current_timestamp - INTERVAL 2 HOURS',
        options => map({', '.join(f"'{k}', '{v}'" for k, v in
compaction_config_streaming.items())})
    )
""")
```

Tables analytiques (requêtes complexes) :

```
# Configuration optimisée pour analytique
compaction_config_analytics = {
```

```
    'target-file-size-bytes': '536870912',  # 512 Mo
    'min-input-files': '5',
    'max-concurrent-file-group-rewrites': '10',
    'rewrite-all': 'false'  # Uniquement fichiers sous-optimaux
}

# Avec tri pour optimisation des requêtes
spark.sql("""
    CALL lakehouse.system.rewrite_data_files(
        table => 'gold.transactions',
        strategy => 'sort',
        sort_order => 'date_transaction DESC, region',
        options => map(
            'target-file-size-bytes', '536870912',
            'min-input-files', '5'
        )
    )
""")
```

Tables historiques (archivage) :

```
# Configuration pour tables peu modifiées
compaction_config_archive = {
    'target-file-size-bytes': '1073741824',  # 1 Go (gros fichiers pour archivage)
    'min-input-files': '10',
    'rewrite-all': 'true'  # Optimisation complète
}
```

Gestion des Snapshots

Expiration des Snapshots

Les snapshots permettent le Time Travel et garantissent les lectures cohérentes, mais leur accumulation consomme de l'espace et ralentit les opérations de métadonnées. L'expiration contrôlée des anciens snapshots équilibre ces besoins.

Procédure d'expiration :

```
from datetime import datetime, timedelta

# Expiration des snapshots plus vieux que 7 jours
retention_date = datetime.now() - timedelta(days=7)
retention_timestamp = retention_date.strftime('%Y-%m-%d %H:%M:%S')

spark.sql(f"""
    CALL lakehouse.system.expire_snapshots(
        table => 'gold.transactions',
        older_than => TIMESTAMP '{retention_timestamp}',
        retain_last => 10,
        max_concurrent_deletes => 50,
        stream_results => true
```

```
    )
""")
```

Paramètres d'expiration :

Paramètre	Description	Recommandation
`older_than`	Timestamp limite de rétention	7-30 jours selon besoins
`retain_last`	Nombre minimum de snapshots conservés	10-100
`max_concurrent_deletes`	Parallélisme de suppression	50-100
`stream_results`	Affichage progressif	true

Politique de rétention par type de table :

Type de table	Rétention recommandée	Justification
Streaming/temps réel	3-7 jours	Données fraîches, volume élevé
Transactionnelle	7-14 jours	Audit court terme
Analytique	30-90 jours	Analyses historiques
Réglementée	365+ jours ou jamais	Conformité légale
Archive	Conservation indéfinie	Historique complet

Expiration avec conservation de snapshots tagués :

```
# Conservation des snapshots de référence (ex: fin de mois)
# D'abord, tagger les snapshots importants
spark.sql("""
    ALTER TABLE lakehouse.gold.transactions
    CREATE TAG `fin_mois_2024_01`
    AS OF VERSION 1234567890
""")

# L'expiration ne supprime pas les snapshots tagués
spark.sql("""
    CALL lakehouse.system.expire_snapshots(
        table => 'gold.transactions',
        older_than => TIMESTAMP '2024-01-01 00:00:00',
        retain_last => 5
    )
    -- Les snapshots tagués sont préservés automatiquement
""")
```

Nettoyage des Fichiers Orphelins

Les fichiers orphelins sont des fichiers de données qui ne sont plus référencés par aucun snapshot actif. Ils résultent de compactions, d'expirations de snapshots ou d'écritures avortées.

Identification des fichiers orphelins :

```
# Listage des fichiers orphelins (dry run)
orphans_df = spark.sql("""
    CALL lakehouse.system.remove_orphan_files(
        table => 'gold.transactions',
        older_than => TIMESTAMP '2024-01-01 00:00:00',
        dry_run => true
    )
""")

print(f"Fichiers orphelins identifiés: {orphans_df.count()}")
orphans_df.show(truncate=False)
```

Suppression des fichiers orphelins :

```
# Suppression effective (avec précautions)
spark.sql("""
    CALL lakehouse.system.remove_orphan_files(
        table => 'gold.transactions',
        older_than => TIMESTAMP '2024-01-08 00:00:00',
        location => 's3://entreprise-lakehouse/gold/transactions',
        dry_run => false,
        max_concurrent_deletes => 100
    )
""")
```

Migration

De : Nettoyage manuel ad-hoc des fichiers orphelins

Vers : Procédure automatisée hebdomadaire avec validation

Stratégie : Toujours exécuter en dry_run d'abord, valider la liste, puis exécuter la suppression. Conserver un délai de sécurité d'au moins 7 jours entre l'expiration des snapshots et le nettoyage des orphelins pour éviter de supprimer des fichiers encore nécessaires à des requêtes en cours.

Script de nettoyage sécurisé :

```
from datetime import datetime, timedelta
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def safe_orphan_cleanup(spark, table: str, dry_run_first: bool = True):
    """Nettoyage sécurisé des fichiers orphelins."""

    # Délai de sécurité: 14 jours après dernier snapshot expiré
    safety_margin = timedelta(days=14)
    cutoff_date = datetime.now() - safety_margin
    cutoff_timestamp = cutoff_date.strftime('%Y-%m-%d %H:%M:%S')

    logger.info(f"Analyse des orphelins pour {table} (avant {cutoff_timestamp})")

    # Phase 1: Dry run
```

```
    orphans = spark.sql(f"""
        CALL lakehouse.system.remove_orphan_files(
            table => '{table}',
            older_than => TIMESTAMP '{cutoff_timestamp}',
            dry_run => true
        )
    """).collect()

    orphan_count = len(orphans)
    total_size = sum(row.file_size_in_bytes for row in orphans) / (1024**3)

    logger.info(f"Fichiers orphelins identifiés: {orphan_count} ({total_size:.2f} Go)")

    if orphan_count == 0:
        logger.info("Aucun fichier orphelin à supprimer")
        return

    if dry_run_first:
        logger.info("Mode dry_run activé - pas de suppression")
        return orphans

    # Phase 2: Suppression effective
    logger.info("Suppression des fichiers orphelins...")
    spark.sql(f"""
        CALL lakehouse.system.remove_orphan_files(
            table => '{table}',
            older_than => TIMESTAMP '{cutoff_timestamp}',
            dry_run => false,
            max_concurrent_deletes => 100
        )
    """)

    logger.info(f"Suppression terminée: {orphan_count} fichiers, {total_size:.2f} Go
libérés")
    return orphan_count, total_size
```

Réécriture des Fichiers de Manifeste

Les fichiers de manifeste peuvent devenir fragmentés ou contenir des références obsolètes. La réécriture optimise leur structure pour accélérer la planification des requêtes.

Réécriture des manifestes :

```
# Réécriture pour consolider les manifestes
spark.sql("""
    CALL lakehouse.system.rewrite_manifests(
        table => 'gold.transactions',
        use_caching => true
    )
""")
```

Quand réécrire les manifestes :

- Après une série de compactions majeures
- Lorsque le nombre de manifestes dépasse 100-200
- Si la planification des requêtes devient lente

- Après des opérations de maintenance importantes

```
# Vérification du nombre de manifestes
manifests_info = spark.sql("""
    SELECT COUNT(*) as manifest_count,
           SUM(added_data_files_count) as total_files,
           AVG(added_data_files_count) as avg_files_per_manifest
    FROM lakehouse.gold.transactions.manifests
""").collect()[0]

if manifests_info.manifest_count > 100:
    logger.info(f"Réécriture recommandée: {manifests_info.manifest_count} manifestes")
    spark.sql("""
        CALL lakehouse.system.rewrite_manifests(
            table => 'gold.transactions'
        )
    """)
```

Monitoring et Observabilité

Métriques Essentielles

Un monitoring efficace du Lakehouse Iceberg repose sur la collecte et l'analyse de métriques couvrant plusieurs dimensions.

Métriques de santé des tables :

```
def collect_table_metrics(spark, table: str) -> dict:
    """Collecte les métriques de santé d'une table Iceberg."""

    # Statistiques des fichiers
    files_stats = spark.sql(f"""
        SELECT
            COUNT(*) as total_files,
            SUM(file_size_in_bytes) / (1024*1024*1024) as total_size_gb,
            AVG(file_size_in_bytes) / (1024*1024) as avg_file_size_mb,
            MIN(file_size_in_bytes) / (1024*1024) as min_file_size_mb,
            MAX(file_size_in_bytes) / (1024*1024) as max_file_size_mb,
            SUM(CASE WHEN file_size_in_bytes < 104857600 THEN 1 ELSE 0 END) as
small_files_count,
            SUM(record_count) as total_records
        FROM {table}.files
    """).collect()[0]

    # Statistiques des snapshots
    snapshots_stats = spark.sql(f"""
        SELECT
            COUNT(*) as snapshot_count,
            MIN(committed_at) as oldest_snapshot,
            MAX(committed_at) as latest_snapshot
        FROM {table}.snapshots
    """).collect()[0]
```

```
    # Statistiques des manifestes
    manifests_stats = spark.sql(f"""
        SELECT
            COUNT(*) as manifest_count,
            SUM(added_data_files_count) as total_manifest_entries
        FROM {table}.manifests
    """).collect()[0]

    # Statistiques des partitions
    partitions_stats = spark.sql(f"""
        SELECT COUNT(DISTINCT partition) as partition_count
        FROM {table}.files
    """).collect()[0]

    return {
        'table': table,
        'total_files': files_stats.total_files,
        'total_size_gb': float(files_stats.total_size_gb or 0),
        'avg_file_size_mb': float(files_stats.avg_file_size_mb or 0),
        'small_files_count': files_stats.small_files_count,
        'small_files_ratio': files_stats.small_files_count / files_stats.total_files if
files_stats.total_files > 0 else 0,
        'total_records': files_stats.total_records,
        'snapshot_count': snapshots_stats.snapshot_count,
        'manifest_count': manifests_stats.manifest_count,
        'partition_count': partitions_stats.partition_count
    }
```

Seuils d'alerte recommandés :

Métrique	Seuil Warning	Seuil Critical	Action
Ratio petits fichiers	> 20%	> 50%	Compaction urgente
Nombre de snapshots	> 100	> 500	Expiration
Nombre de manifestes	> 100	> 500	Réécriture
Taille moyenne fichier	< 100 Mo	< 50 Mo	Compaction
Fichiers orphelins (Go)	> 100	> 500	Nettoyage

Dashboard Grafana pour Lakehouse

Configuration Prometheus pour métriques Iceberg :

```
# prometheus-iceberg-rules.yaml
groups:
  - name: iceberg-table-health
    interval: 5m
    rules:
      # Métrique personnalisée: ratio de petits fichiers
      - record: iceberg_small_files_ratio
        expr: |
          iceberg_table_small_files_count / iceberg_table_total_files
        labels:
```

```
          severity: "{{ if gt .Value 0.5 }}critical{{ else if gt .Value 0.2 }}
warning{{ else }}ok{{ end }}"

      # Alerte: trop de petits fichiers
      - alert: IcebergSmallFilesHigh
        expr: iceberg_small_files_ratio > 0.2
        for: 30m
        labels:
          severity: warning
        annotations:
          summary: "Table {{ $labels.table }} a {{ $value | humanizePercentage }} de
petits fichiers"
          description: "Compaction recommandée pour améliorer les performances"

      # Alerte: snapshots accumulés
      - alert: IcebergSnapshotsAccumulated
        expr: iceberg_table_snapshot_count > 100
        for: 1h
        labels:
          severity: warning
        annotations:
          summary: "Table {{ $labels.table }} a {{ $value }} snapshots"
          description: "Expiration des snapshots recommandée"

      # Alerte: croissance anormale
      - alert: IcebergAbnormalGrowth
        expr: |
          (iceberg_table_size_bytes - iceberg_table_size_bytes offset 1d)
          / iceberg_table_size_bytes offset 1d > 0.5
        for: 1h
        labels:
          severity: warning
        annotations:
          summary: "Croissance anormale de {{ $labels.table }}"
          description: "La table a grandi de plus de 50% en 24h"
```

Dashboard Grafana :

```
{
  "dashboard": {
    "title": "Lakehouse Iceberg - Santé des Tables",
    "panels": [
      {
        "title": "Vue d'ensemble des tables",
        "type": "table",
        "targets": [
          {
            "expr": "iceberg_table_total_size_gb",
            "legendFormat": "{{ table }}"
          }
        ],
        "fieldConfig": {
          "defaults": {
            "thresholds": {
              "mode": "absolute",
              "steps": [
                {"color": "green", "value": null},
                {"color": "yellow", "value": 100},
```

```
                {"color": "red", "value": 500}
              ]
            }
          }
        }
      },
      {
        "title": "Ratio de petits fichiers par table",
        "type": "gauge",
        "targets": [
          {
            "expr": "iceberg_small_files_ratio",
            "legendFormat": "{{ table }}"
          }
        ],
        "fieldConfig": {
          "defaults": {
            "max": 1,
            "thresholds": {
              "mode": "percentage",
              "steps": [
                {"color": "green", "value": null},
                {"color": "yellow", "value": 20},
                {"color": "red", "value": 50}
              ]
            }
          }
        }
      },
      {
        "title": "Évolution du nombre de fichiers",
        "type": "timeseries",
        "targets": [
          {
            "expr": "iceberg_table_total_files",
            "legendFormat": "{{ table }}"
          }
        ]
      },
      {
        "title": "Opérations de maintenance",
        "type": "stat",
        "targets": [
          {
            "expr": "sum(rate(iceberg_compaction_completed_total[24h]))",
            "legendFormat": "Compactions/24h"
          },
          {
            "expr": "sum(rate(iceberg_snapshots_expired_total[24h]))",
            "legendFormat": "Expirations/24h"
          }
        ]
      }
    ]
  }
}
```

Collecte Automatisée des Métriques

Service de collecte des métriques :

```
from prometheus_client import Gauge, start_http_server
import schedule
import time

# Définition des métriques Prometheus
iceberg_total_files = Gauge(
    'iceberg_table_total_files',
    'Nombre total de fichiers',
    ['catalog', 'database', 'table']
)

iceberg_total_size = Gauge(
    'iceberg_table_total_size_gb',
    'Taille totale en Go',
    ['catalog', 'database', 'table']
)

iceberg_small_files = Gauge(
    'iceberg_table_small_files_count',
    'Nombre de petits fichiers (<100Mo)',
    ['catalog', 'database', 'table']
)

iceberg_snapshot_count = Gauge(
    'iceberg_table_snapshot_count',
    'Nombre de snapshots',
    ['catalog', 'database', 'table']
)

iceberg_avg_file_size = Gauge(
    'iceberg_table_avg_file_size_mb',
    'Taille moyenne des fichiers en Mo',
    ['catalog', 'database', 'table']
)

class IcebergMetricsCollector:
    def __init__(self, spark, catalog: str, tables: list):
        self.spark = spark
        self.catalog = catalog
        self.tables = tables

    def collect_all_metrics(self):
        """Collecte les métriques pour toutes les tables configurées."""
        for table_path in self.tables:
            try:
                self._collect_table_metrics(table_path)
            except Exception as e:
                logger.error(f"Erreur collecte métriques {table_path}: {e}")

    def _collect_table_metrics(self, table_path: str):
        """Collecte les métriques pour une table."""
        parts = table_path.split('.')
        database = parts[-2] if len(parts) > 1 else 'default'
        table = parts[-1]
```

```
        metrics = collect_table_metrics(self.spark, table_path)

        labels = [self.catalog, database, table]

        iceberg_total_files.labels(*labels).set(metrics['total_files'])
        iceberg_total_size.labels(*labels).set(metrics['total_size_gb'])
        iceberg_small_files.labels(*labels).set(metrics['small_files_count'])
        iceberg_snapshot_count.labels(*labels).set(metrics['snapshot_count'])
        iceberg_avg_file_size.labels(*labels).set(metrics['avg_file_size_mb'])

# Démarrage du serveur Prometheus
start_http_server(8000)

# Configuration de la collecte périodique
collector = IcebergMetricsCollector(
    spark=spark,
    catalog='lakehouse',
    tables=[
        'lakehouse.gold.transactions',
        'lakehouse.gold.clients',
        'lakehouse.silver.events',
        'lakehouse.bronze.raw_data'
    ]
)

schedule.every(5).minutes.do(collector.collect_all_metrics)

while True:
    schedule.run_pending()
    time.sleep(1)
```

Automatisation des Opérations

Orchestration avec Apache Airflow

L'automatisation des opérations de maintenance via Airflow garantit leur exécution régulière et fiable.

DAG de maintenance quotidienne :

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.providers.apache.spark.operators.spark_submit import SparkSubmitOperator
from airflow.operators.email import EmailOperator
from airflow.utils.task_group import TaskGroup
from datetime import datetime, timedelta

default_args = {
    'owner': 'data-platform',
    'depends_on_past': False,
    'email': ['data-ops@entreprise.ca'],
    'email_on_failure': True,
    'retries': 2,
    'retry_delay': timedelta(minutes=15),
```

```
    'retry_exponential_backoff': True
}

# Liste des tables à maintenir
TABLES_CRITIQUES = [
    'lakehouse.gold.transactions',
    'lakehouse.gold.clients',
    'lakehouse.silver.events'
]

TABLES_STANDARD = [
    'lakehouse.bronze.raw_events',
    'lakehouse.bronze.raw_logs'
]

with DAG(
    'lakehouse_maintenance_quotidienne',
    default_args=default_args,
    description='Maintenance quotidienne du Lakehouse Iceberg',
    schedule_interval='0 3 * * *',  # 3h00 quotidien
    start_date=datetime(2024, 1, 1),
    catchup=False,
    tags=['maintenance', 'iceberg', 'production'],
    max_active_runs=1
) as dag:

    # Collecte des métriques pré-maintenance
    collect_metrics_pre = SparkSubmitOperator(
        task_id='collect_metrics_pre',
        application='s3://lakehouse/jobs/collect_metrics.py',
        conf={'spark.app.name': 'MetricsPreMaintenance'},
        application_args=['--output', 's3://lakehouse/metrics/pre/{{ ds }}']
    )

    # Groupe: Compaction des tables critiques
    with TaskGroup('compaction_critiques') as compaction_critiques:
        for table in TABLES_CRITIQUES:
            table_name = table.split('.')[-1]
            SparkSubmitOperator(
                task_id=f'compact_{table_name}',
                application='s3://lakehouse/jobs/compaction.py',
                conf={
                    'spark.app.name': f'Compaction-{table_name}',
                    'spark.executor.instances': '10',
                    'spark.executor.memory': '8g'
                },
                application_args=[
                    '--table', table,
                    '--strategy', 'binpack',
                    '--target-size', '512MB'
                ]
            )

    # Groupe: Compaction des tables standard
    with TaskGroup('compaction_standard') as compaction_standard:
        for table in TABLES_STANDARD:
            table_name = table.split('.')[-1]
            SparkSubmitOperator(
                task_id=f'compact_{table_name}',
```

```
            application='s3://lakehouse/jobs/compaction.py',
            conf={
                'spark.app.name': f'Compaction-{table_name}',
                'spark.executor.instances': '5'
            },
            application_args=[
                '--table', table,
                '--strategy', 'binpack',
                '--target-size', '512MB',
                '--partitions-last-days', '7'
            ]
        )

# Expiration des snapshots
with TaskGroup('expiration_snapshots') as expiration_snapshots:
    for table in TABLES_CRITIQUES + TABLES_STANDARD:
        table_name = table.split('.')[-1]
        SparkSubmitOperator(
            task_id=f'expire_{table_name}',
            application='s3://lakehouse/jobs/expire_snapshots.py',
            application_args=[
                '--table', table,
                '--retention-days', '7',
                '--retain-last', '10'
            ]
        )

# Nettoyage des orphelins (une fois par semaine le dimanche)
cleanup_orphans = SparkSubmitOperator(
    task_id='cleanup_orphans',
    application='s3://lakehouse/jobs/cleanup_orphans.py',
    application_args=['--older-than-days', '14'],
    trigger_rule='all_done'  # Exécute même si compaction échoue
)

# Collecte des métriques post-maintenance
collect_metrics_post = SparkSubmitOperator(
    task_id='collect_metrics_post',
    application='s3://lakehouse/jobs/collect_metrics.py',
    application_args=['--output', 's3://lakehouse/metrics/post/{{ ds }}']
)

# Génération du rapport
generate_report = PythonOperator(
    task_id='generate_report',
    python_callable=generate_maintenance_report,
    op_kwargs={'date': '{{ ds }}'}
)

# Notification
notify_success = EmailOperator(
    task_id='notify_success',
    to='data-ops@entreprise.ca',
    subject='[Lakehouse] Maintenance quotidienne terminée - {{ ds }}',
    html_content='{{ task_instance.xcom_pull(task_ids="generate_report") }}'
)

# Dépendances
collect_metrics_pre >> [compaction_critiques, compaction_standard]
```

```
    compaction_critiques >> expiration_snapshots
    compaction_standard >> expiration_snapshots
    expiration_snapshots >> cleanup_orphans >> collect_metrics_post
    collect_metrics_post >> generate_report >> notify_success
```

Script de compaction réutilisable :

```
# jobs/compaction.py
import argparse
from pyspark.sql import SparkSession
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def main():
    parser = argparse.ArgumentParser(description='Compaction Iceberg')
    parser.add_argument('--table', required=True, help='Table à compacter')
    parser.add_argument('--strategy', default='binpack', choices=['binpack', 'sort'])
    parser.add_argument('--target-size', default='512MB', help='Taille cible des
fichiers')
    parser.add_argument('--partitions-last-days', type=int, help='Limiter aux N derniers
jours')
    parser.add_argument('--sort-order', help='Ordre de tri (pour strategy=sort)')
    args = parser.parse_args()

    spark = SparkSession.builder \
        .appName(f"Compaction-{args.table}") \
        .getOrCreate()

    # Conversion de la taille cible
    size_bytes = parse_size(args.target_size)

    # Construction de la requête
    options = {
        'target-file-size-bytes': str(size_bytes),
        'min-input-files': '3',
        'partial-progress.enabled': 'true'
    }

    where_clause = ""
    if args.partitions_last_days:
        where_clause = f"WHERE date_partition >= current_date - INTERVAL
'{args.partitions_last_days}' DAY"

    sort_order = ""
    if args.strategy == 'sort' and args.sort_order:
        sort_order = f", sort_order => '{args.sort_order}'"

    query = f"""
        CALL lakehouse.system.rewrite_data_files(
            table => '{args.table}',
            strategy => '{args.strategy}'
            {sort_order},
            options => map({', '.join(f"'{k}', '{v}'" for k, v in options.items())})
        )
    """
```

```
    logger.info(f"Exécution compaction: {args.table}")
    logger.info(f"Stratégie: {args.strategy}, Taille cible: {args.target_size}")

    result = spark.sql(query).collect()

    rewritten_files = sum(r.rewritten_data_files_count for r in result)
    added_files = sum(r.added_data_files_count for r in result)

    logger.info(f"Compaction terminée: {rewritten_files} fichiers réécrits → {added_files}
fichiers")

    spark.stop()

def parse_size(size_str: str) -> int:
    """Convertit une chaîne de taille en bytes."""
    units = {'B': 1, 'KB': 1024, 'MB': 1024**2, 'GB': 1024**3}
    size_str = size_str.upper().strip()
    for unit, multiplier in units.items():
        if size_str.endswith(unit):
            return int(float(size_str[:-len(unit)]) * multiplier)
    return int(size_str)

if __name__ == '__main__':
    main()
```

Maintenance Événementielle

Au-delà de la maintenance planifiée, certaines opérations doivent se déclencher sur événements.

Triggers automatiques :

```
from airflow.sensors.external_task import ExternalTaskSensor
from airflow.operators.trigger_dagrun import TriggerDagRunOperator

# DAG de compaction déclenchée par accumulation
with DAG(
    'lakehouse_compaction_triggered',
    schedule_interval=None,  # Déclenché par événement
    catchup=False
) as dag:

    def check_compaction_needed(**context):
        """Vérifie si une compaction est nécessaire."""
        table = context['dag_run'].conf.get('table')
        metrics = collect_table_metrics(spark, table)

        # Critères de déclenchement
        if metrics['small_files_ratio'] > 0.3:
            return True
        if metrics['total_files'] > 10000:
            return True

        return False

    check_needed = PythonOperator(
        task_id='check_compaction_needed',
        python_callable=check_compaction_needed
```

```
    )

    run_compaction = SparkSubmitOperator(
        task_id='run_compaction',
        application='s3://lakehouse/jobs/compaction.py',
        application_args=[
            '--table', '{{ dag_run.conf.table }}',
            '--strategy', 'binpack'
        ]
    )

    check_needed >> run_compaction

# Fonction pour déclencher depuis un monitoring externe
def trigger_compaction_if_needed(table: str, metrics: dict):
    """Déclenche une compaction si les seuils sont dépassés."""
    from airflow.api.common.experimental.trigger_dag import trigger_dag

    if metrics['small_files_ratio'] > 0.3:
        trigger_dag(
            dag_id='lakehouse_compaction_triggered',
            conf={'table': table}
        )
```

Gestion des Incidents

Procédures de Diagnostic

Runbook de diagnostic des problèmes de performance :

```
# Runbook: Diagnostic Performance Lakehouse

## Symptôme: Requêtes lentes

### Étape 1: Vérifier l'état des tables concernées
```sql
-- Statistiques des fichiers
SELECT
    COUNT(*) as total_files,
    AVG(file_size_in_bytes)/(1024*1024) as avg_size_mb,
    SUM(CASE WHEN file_size_in_bytes < 104857600 THEN 1 ELSE 0 END) as small_files
FROM lakehouse.gold.transactions.files;
```

Étape 2: Vérifier les statistiques de table

```
-- Dernière mise à jour des stats
SELECT * FROM lakehouse.gold.transactions.metadata_log_entries
ORDER BY timestamp DESC LIMIT 5;
```
```

Étape 3: Analyser le plan de requête

```
EXPLAIN ANALYZE SELECT ... FROM lakehouse.gold.transactions WHERE ...;
```

Actions correctives:

- Si small_files > 20% du total → Exécuter compaction
- Si statistiques obsolètes → ANALYZE TABLE
- Si plan de requête inefficace → Vérifier partitionnement

```
**Script de diagnostic automatisé** :

```python
class LakehouseDiagnostics:
    def __init__(self, spark):
        self.spark = spark

    def diagnose_table(self, table: str) -> dict:
        """Diagnostic complet d'une table."""
        issues = []
        recommendations = []

        # 1. Analyse des fichiers
        files_analysis = self._analyze_files(table)
        if files_analysis['small_files_ratio'] > 0.2:
            issues.append(f"Trop de petits fichiers:
{files_analysis['small_files_ratio']:.1%}")
            recommendations.append("Exécuter compaction binpack")

        # 2. Analyse des snapshots
        snapshots_analysis = self._analyze_snapshots(table)
        if snapshots_analysis['count'] > 100:
            issues.append(f"Accumulation de snapshots: {snapshots_analysis['count']}")
            recommendations.append("Exécuter expiration des snapshots")

        # 3. Analyse des manifestes
        manifests_analysis = self._analyze_manifests(table)
        if manifests_analysis['count'] > 100:
            issues.append(f"Fragmentation des manifestes: {manifests_analysis['count']}")
            recommendations.append("Réécrire les manifestes")

        # 4. Vérification des statistiques
        stats_analysis = self._analyze_statistics(table)
        if stats_analysis['age_days'] > 7:
            issues.append(f"Statistiques obsolètes: {stats_analysis['age_days']} jours")
            recommendations.append("Mettre à jour les statistiques (ANALYZE TABLE)")

        return {
            'table': table,
            'health_score': self._calculate_health_score(files_analysis,
snapshots_analysis, manifests_analysis),
            'issues': issues,
            'recommendations': recommendations,
            'details': {
                'files': files_analysis,
```
```

```
                'snapshots': snapshots_analysis,
                'manifests': manifests_analysis,
                'statistics': stats_analysis
            }
        }

    def _analyze_files(self, table: str) -> dict:
        result = self.spark.sql(f"""
            SELECT
                COUNT(*) as total,
                SUM(file_size_in_bytes) as total_size,
                AVG(file_size_in_bytes) as avg_size,
                SUM(CASE WHEN file_size_in_bytes < 104857600 THEN 1 ELSE 0 END) as
small_count
            FROM {table}.files
        """).collect()[0]

        return {
            'total_files': result.total,
            'total_size_gb': result.total_size / (1024**3) if result.total_size else 0,
            'avg_size_mb': result.avg_size / (1024**2) if result.avg_size else 0,
            'small_files_count': result.small_count,
            'small_files_ratio': result.small_count / result.total if result.total > 0
else 0
        }

    def _calculate_health_score(self, files, snapshots, manifests) -> int:
        """Calcule un score de santé de 0 à 100."""
        score = 100

        # Pénalités pour petits fichiers
        if files['small_files_ratio'] > 0.5:
            score -= 30
        elif files['small_files_ratio'] > 0.2:
            score -= 15

        # Pénalités pour snapshots
        if snapshots['count'] > 500:
            score -= 20
        elif snapshots['count'] > 100:
            score -= 10

        # Pénalités pour manifestes
        if manifests['count'] > 500:
            score -= 20
        elif manifests['count'] > 100:
            score -= 10

        return max(0, score)
```

Récupération après Incident

Procédure de rollback après écriture corrompue :

```
def rollback_to_snapshot(spark, table: str, snapshot_id: int = None,
                         timestamp: str = None):
    """Rollback d'une table vers un état antérieur."""
```

```
    if snapshot_id:
        # Rollback vers snapshot spécifique
        spark.sql(f"""
            CALL lakehouse.system.rollback_to_snapshot(
                table => '{table}',
                snapshot_id => {snapshot_id}
            )
        """)
        logger.info(f"Rollback vers snapshot {snapshot_id} effectué")

    elif timestamp:
        # Rollback vers timestamp
        spark.sql(f"""
            CALL lakehouse.system.rollback_to_timestamp(
                table => '{table}',
                timestamp => TIMESTAMP '{timestamp}'
            )
        """)
        logger.info(f"Rollback vers {timestamp} effectué")

    else:
        # Rollback vers snapshot précédent
        snapshots = spark.sql(f"""
            SELECT snapshot_id, committed_at
            FROM {table}.snapshots
            ORDER BY committed_at DESC
            LIMIT 2
        """).collect()

        if len(snapshots) < 2:
            raise ValueError("Pas assez de snapshots pour rollback")

        previous_snapshot = snapshots[1].snapshot_id
        spark.sql(f"""
            CALL lakehouse.system.rollback_to_snapshot(
                table => '{table}',
                snapshot_id => {previous_snapshot}
            )
        """)
        logger.info(f"Rollback vers snapshot précédent {previous_snapshot}")
```

Récupération de métadonnées corrompues :

```
def recover_from_metadata_corruption(spark, table: str,
                                     metadata_backup_path: str):
    """Récupération depuis une sauvegarde de métadonnées."""

    # 1. Identifier le dernier fichier metadata valide
    valid_metadata = find_valid_metadata(metadata_backup_path)

    if not valid_metadata:
        raise ValueError("Aucun fichier metadata valide trouvé")

    # 2. Réenregistrer la table avec le metadata valide
    catalog = table.split('.')[0]
    db_table = '.'.join(table.split('.')[1:])
```

```
    # Supprimer l'entrée catalogue corrompue
    spark.sql(f"DROP TABLE IF EXISTS {table}")

    # Réenregistrer avec metadata valide
    spark.sql(f"""
        CALL {catalog}.system.register_table(
            table => '{db_table}',
            metadata_file => '{valid_metadata}'
        )
    """)

    logger.info(f"Table {table} récupérée depuis {valid_metadata}")

def find_valid_metadata(backup_path: str) -> str:
    """Trouve le fichier metadata le plus récent valide."""
    import json
    from smart_open import open

    # Lister les fichiers metadata dans le backup
    metadata_files = list_files(backup_path, pattern='*.metadata.json')

    # Trier par date décroissante
    metadata_files.sort(reverse=True)

    for metadata_file in metadata_files:
        try:
            # Vérifier que le fichier est valide
            with open(metadata_file, 'r') as f:
                metadata = json.load(f)

            # Vérifications de base
            if 'format-version' in metadata and 'snapshots' in metadata:
                return metadata_file

        except Exception as e:
            logger.warning(f"Fichier metadata invalide: {metadata_file} - {e}")
            continue

    return None
```

Sauvegarde et Récupération

Stratégies de Sauvegarde

Sauvegarde des métadonnées :

Les fichiers de métadonnées Iceberg sont critiques — leur perte signifie la perte d'accès aux données. Une stratégie de sauvegarde robuste est essentielle.

```
import boto3
from datetime import datetime
```

```python
class IcebergBackupManager:
    def __init__(self, source_bucket: str, backup_bucket: str):
        self.s3 = boto3.client('s3')
        self.source_bucket = source_bucket
        self.backup_bucket = backup_bucket

    def backup_table_metadata(self, table_path: str):
        """Sauvegarde les métadonnées d'une table."""

        metadata_prefix = f"{table_path}/metadata/"
        backup_date = datetime.now().strftime('%Y-%m-%d_%H%M%S')
        backup_prefix = f"backups/{backup_date}/{table_path}/metadata/"

        # Lister tous les fichiers metadata
        paginator = self.s3.get_paginator('list_objects_v2')

        copied_files = 0
        for page in paginator.paginate(Bucket=self.source_bucket,
                                       Prefix=metadata_prefix):
            for obj in page.get('Contents', []):
                source_key = obj['Key']
                dest_key = source_key.replace(metadata_prefix, backup_prefix)

                # Copie vers bucket de backup
                self.s3.copy_object(
                    Bucket=self.backup_bucket,
                    Key=dest_key,
                    CopySource={'Bucket': self.source_bucket, 'Key': source_key}
                )
                copied_files += 1

        logger.info(f"Sauvegarde {table_path}: {copied_files} fichiers")
        return backup_prefix

    def backup_all_tables(self, tables: list):
        """Sauvegarde toutes les tables configurées."""
        backup_manifest = {
            'timestamp': datetime.now().isoformat(),
            'tables': {}
        }

        for table in tables:
            table_path = table.replace('.', '/')
            backup_path = self.backup_table_metadata(table_path)
            backup_manifest['tables'][table] = backup_path

        # Sauvegarder le manifeste
        manifest_key = f"backups/{datetime.now().strftime('%Y-%m-%d_%H%M%S')}/
manifest.json"
        self.s3.put_object(
            Bucket=self.backup_bucket,
            Key=manifest_key,
            Body=json.dumps(backup_manifest)
        )

        return backup_manifest
```

Configuration de la réplication S3 :

```
# terraform/s3-replication.tf

resource "aws_s3_bucket_replication_configuration" "lakehouse_replication" {
  bucket = aws_s3_bucket.lakehouse.id
  role   = aws_iam_role.replication.arn

  rule {
    id     = "metadata-replication"
    status = "Enabled"

    filter {
      prefix = "warehouse/"
    }

    destination {
      bucket        = aws_s3_bucket.lakehouse_backup.arn
      storage_class = "STANDARD_IA"

      replication_time {
        status = "Enabled"
        time {
          minutes = 15
        }
      }

      metrics {
        status = "Enabled"
        event_threshold {
          minutes = 15
        }
      }
    }

    delete_marker_replication {
      status = "Disabled"  # Conserver les données supprimées dans le backup
    }
  }
}
```

Plan de Reprise après Sinistre

RTO/RPO par criticité :

Niveau	Tables	RPO	RTO	Stratégie
Critique	Transactions, Clients	1h	4h	Réplication temps réel
Élevé	Événements, Logs	4h	8h	Backup horaire
Standard	Référentiels	24h	24h	Backup quotidien
Faible	Archives	7j	48h	Backup hebdomadaire

Procédure de reprise :

```python
class DisasterRecovery:
    def __init__(self, primary_region: str, dr_region: str):
        self.primary_region = primary_region
        self.dr_region = dr_region
        self.s3_primary = boto3.client('s3', region_name=primary_region)
        self.s3_dr = boto3.client('s3', region_name=dr_region)

    def initiate_failover(self, tables: list):
        """Initie le basculement vers la région DR."""

        failover_report = {
            'initiated_at': datetime.now().isoformat(),
            'tables': []
        }

        for table in tables:
            try:
                # 1. Vérifier la disponibilité dans DR
                dr_status = self._check_dr_availability(table)

                # 2. Mettre à jour le catalogue pour pointer vers DR
                self._update_catalog_to_dr(table)

                # 3. Valider l'accès aux données
                validation = self._validate_table_access(table)

                failover_report['tables'].append({
                    'table': table,
                    'status': 'success',
                    'dr_snapshot': dr_status['latest_snapshot'],
                    'data_loss_minutes': dr_status['lag_minutes']
                })

            except Exception as e:
                failover_report['tables'].append({
                    'table': table,
                    'status': 'failed',
                    'error': str(e)
                })

        return failover_report

    def _check_dr_availability(self, table: str) -> dict:
        """Vérifie la disponibilité et la fraîcheur des données DR."""
        # Implémentation spécifique selon l'architecture
        pass

    def _update_catalog_to_dr(self, table: str):
        """Met à jour le catalogue pour pointer vers la région DR."""
        # Mise à jour des références dans le catalogue REST
        pass
```

Optimisation Continue

Analyse des Patterns de Requêtes

L'optimisation continue repose sur l'analyse des patterns d'utilisation réels pour guider les décisions de maintenance.

```
class QueryPatternAnalyzer:
    def __init__(self, spark):
        self.spark = spark

    def analyze_query_patterns(self, table: str, days: int = 30) -> dict:
        """Analyse les patterns de requêtes sur une table."""

        # Récupération des logs de requêtes (depuis Trino/système d'audit)
        query_logs = self.spark.sql(f"""
            SELECT
                query_text,
                execution_time_ms,
                rows_read,
                bytes_read,
                query_timestamp
            FROM audit.query_logs
            WHERE table_name = '{table}'
              AND query_timestamp >= current_date - INTERVAL '{days}' DAY
        """)

        # Analyse des colonnes filtrées
        filter_columns = self._extract_filter_columns(query_logs)

        # Analyse des colonnes projetées
        projected_columns = self._extract_projected_columns(query_logs)

        # Analyse des patterns temporels
        temporal_patterns = self._analyze_temporal_patterns(query_logs)

        return {
            'table': table,
            'total_queries': query_logs.count(),
            'most_filtered_columns': filter_columns[:5],
            'most_projected_columns': projected_columns[:10],
            'peak_hours': temporal_patterns['peak_hours'],
            'recommendations': self._generate_recommendations(
                filter_columns, projected_columns
            )
        }

    def _generate_recommendations(self, filter_cols, project_cols) -> list:
        """Génère des recommandations d'optimisation."""
        recommendations = []

        # Recommandation de tri basée sur les filtres fréquents
        if filter_cols:
            top_filter = filter_cols[0]
            recommendations.append({
                'type': 'sort_order',
                'description': f"Considérer un tri par {top_filter['column']}",
```

```
                'impact': 'Amélioration potentielle des scans filtrés',
                'effort': 'moyen'
            })

        # Recommandation de Z-order si plusieurs colonnes filtrées
        if len(filter_cols) >= 2:
            cols = [c['column'] for c in filter_cols[:3]]
            recommendations.append({
                'type': 'z_order',
                'description': f"Considérer Z-order sur {', '.join(cols)}",
                'impact': 'Amélioration des requêtes multi-colonnes',
                'effort': 'élevé'
            })

        return recommendations
```

Ajustement Automatique des Paramètres

```
class AdaptiveMaintenanceScheduler:
    """Ajuste automatiquement les paramètres de maintenance selon les métriques."""

    def __init__(self, spark):
        self.spark = spark
        self.default_config = {
            'compaction_threshold': 0.2,  # Ratio petits fichiers
            'snapshot_retention_days': 7,
            'compaction_frequency_hours': 24
        }

    def calculate_optimal_config(self, table: str) -> dict:
        """Calcule la configuration optimale basée sur les métriques."""

        metrics = collect_table_metrics(self.spark, table)
        query_patterns = QueryPatternAnalyzer(self.spark).analyze_query_patterns(table)

        config = self.default_config.copy()

        # Ajustement basé sur le taux d'écriture
        write_rate = self._estimate_write_rate(table)
        if write_rate > 1000:  # Plus de 1000 écritures/heure
            config['compaction_frequency_hours'] = 1
            config['compaction_threshold'] = 0.1
        elif write_rate > 100:
            config['compaction_frequency_hours'] = 6
            config['compaction_threshold'] = 0.15

        # Ajustement basé sur la taille de la table
        if metrics['total_size_gb'] > 1000:  # Plus de 1 To
            config['snapshot_retention_days'] = 3  # Rétention réduite

        # Ajustement basé sur la fréquence des requêtes
        if query_patterns['total_queries'] > 10000:  # Table très consultée
            config['compaction_threshold'] = 0.1  # Compaction plus agressive

        return config
```

```
    def _estimate_write_rate(self, table: str) -> float:
        """Estime le taux d'écriture basé sur l'historique des snapshots."""
        result = self.spark.sql(f"""
            SELECT
                COUNT(*) as snapshot_count,
                TIMESTAMPDIFF(HOUR, MIN(committed_at), MAX(committed_at)) as hours_span
            FROM {table}.snapshots
            WHERE committed_at >= current_timestamp - INTERVAL '24' HOUR
        """).collect()[0]

        if result.hours_span and result.hours_span > 0:
            return result.snapshot_count / result.hours_span
        return 0
```

Études de Cas Canadiennes

Secteur Financier : Banque d'Investissement

Étude de cas : Banque d'investissement canadienne

Secteur : Services financiers - marchés des capitaux

Défi : Maintenir un Lakehouse de 500 To contenant 10 ans d'historique de transactions de marché avec des exigences de rétention réglementaire strictes (7 ans minimum) et des SLA de performance pour les requêtes analytiques (< 30 secondes).

Solution : Architecture de maintenance à trois niveaux avec orchestration Airflow, monitoring Prometheus/Grafana et procédures de récupération automatisées. Stratégie de tiering automatique déplaçant les données > 1 an vers S3 Glacier.

Configuration de maintenance :

```
Tables critiques (données < 90 jours):
- Compaction: toutes les 2 heures
- Taille cible: 256 Mo
- Expiration snapshots: 3 jours, retain_last=50

Tables historiques (données 90 jours - 1 an):
- Compaction: quotidienne
- Taille cible: 512 Mo
- Expiration snapshots: 7 jours, retain_last=10

Tables archives (> 1 an):
- Compaction: mensuelle
- Taille cible: 1 Go
- Expiration snapshots: 30 jours, retain_last=5
```

Résultats :

- Réduction des coûts de stockage de 45% via tiering
- 99.9% de disponibilité sur 2 ans
- Temps de requête P95 < 25 secondes
- Conformité réglementaire maintenue avec audit complet

Secteur Télécommunications : Opérateur Mobile

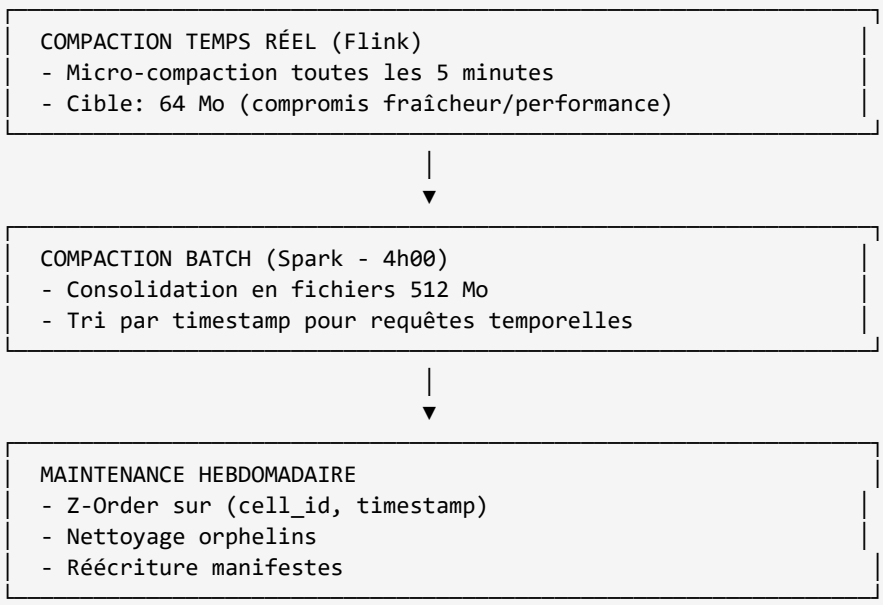
Étude de cas : Opérateur mobile pancanadien

Secteur : Télécommunications

Défi : Gérer l'ingestion de 50 milliards d'événements réseau par jour avec compaction temps réel pour maintenir des performances de requête acceptables pour le NOC (Network Operations Center).

Solution : Architecture de maintenance continue avec compaction streaming intégrée au pipeline Flink, complétée par une compaction batch quotidienne pour optimisation finale.

Architecture de maintenance :



Résultats :

- Latence de requête NOC < 5 secondes malgré 50B événements/jour
- Ratio petits fichiers maintenu < 5%
- Coût de maintenance: 3% du coût total d'infrastructure
- Zéro incident de performance en production

Secteur Commerce : Détaillant Alimentaire

Étude de cas : Chaîne d'épiceries québécoise

Secteur : Commerce de détail alimentaire

Défi : Maintenir un Lakehouse de 50 To avec des ressources d'équipe limitées (2 data engineers) tout en supportant 200 utilisateurs analytiques.

Solution : Automatisation complète de la maintenance via Airflow avec alertes proactives. Utilisation de Dremio pour la couche de requête avec Reflections réduisant la dépendance à une compaction ultra-optimisée.

Stratégie low-touch :

```
# Configuration simplifiée pour équipe réduite
maintenance_config = {
    'compaction': {
        'schedule': 'daily_3am',
        'strategy': 'binpack',  # Simple, pas de tri complexe
        'target_size': '512MB',
        'auto_trigger_threshold': 0.3  # 30% petits fichiers
    },
    'expiration': {
        'schedule': 'daily_4am',
        'retention_days': 14,
        'retain_last': 20
    },
    'orphan_cleanup': {
        'schedule': 'weekly_sunday_5am',
        'safety_margin_days': 21
    },
    'alerting': {
        'small_files_warning': 0.25,
        'small_files_critical': 0.5,
        'notify': 'slack://data-alerts'
    }
}
```

Résultats :

- Temps de maintenance: 2h/semaine (vs 15h/semaine avant automatisation)
- Aucune intervention manuelle requise en 6 mois
- Performance stable malgré croissance de 30% du volume
- ROI de l'automatisation: 200% en première année

Secteur Public : Agence Statistique

Étude de cas : Agence fédérale de statistiques

Secteur : Gouvernement fédéral

Défi : Maintenir un Lakehouse contenant des données de recensement avec exigences de conservation perpétuelle et capacité de reproduire exactement les analyses publiées des décennies plus tard.

Solution : Stratégie de versionnement strict avec tags Iceberg pour chaque publication officielle, jamais d'expiration de snapshots tagués, et archivage géographiquement distribué.

Politique de rétention :

```
-- Aucune expiration automatique pour tables de publication
ALTER TABLE lakehouse.recensement.donnees_2021
SET TBLPROPERTIES ('gc.enabled' = 'false');

-- Tags pour chaque publication officielle
ALTER TABLE lakehouse.recensement.donnees_2021
CREATE TAG `publication_officielle_2022_02_09`
AS OF VERSION 1234567890;

-- Les tags sont JAMAIS supprimés
-- La compaction préserve les fichiers référencés par les tags
```

Résultats :

- Reproductibilité parfaite des analyses sur 20+ ans
- Conformité aux exigences archivistiques fédérales
- Capacité Time Travel illimitée pour données officielles
- Coût de stockage maîtrisé via tiering Glacier pour données non-taggées

---

Bonnes Pratiques et Recommandations

Checklist de Maintenance

Quotidien :

- ☐ Vérification des alertes de monitoring
- ☐ Exécution de la compaction des tables critiques
- ☐ Expiration des snapshots selon politique
- ☐ Vérification des jobs de maintenance planifiés

Hebdomadaire :

- ☐ Nettoyage des fichiers orphelins
- ☐ Revue des métriques de santé des tables
- ☐ Analyse des tendances de croissance
- ☐ Vérification des sauvegardes

Mensuel :

- ☐ Réécriture des manifestes si nécessaire
- ☐ Analyse des patterns de requêtes

- ☐ Ajustement des paramètres de maintenance
- ☐ Test de récupération après sinistre
- ☐ Revue de la politique de rétention

Trimestriel :

- ☐ Audit complet de la santé du Lakehouse
- ☐ Optimisation du tri (Z-order) si applicable
- ☐ Revue de l'architecture de maintenance
- ☐ Mise à jour de la documentation

Matrice de Décision

Fréquence de compaction :

Taux d'écriture	Volume table	Fréquence recommandée
Streaming continu	> 100 Go/jour	Horaire
Batch fréquent	10-100 Go/jour	Toutes les 6h
Batch quotidien	1-10 Go/jour	Quotidienne
Batch hebdomadaire	< 1 Go/jour	Hebdomadaire

Rétention des snapshots :

Type de table	Fréquence modification	Rétention recommandée
Temps réel	Continue	3-7 jours
Transactionnelle	Quotidienne	7-14 jours
Analytique	Hebdomadaire	30-90 jours
Réglementée	Variable	Selon exigences légales

Anti-Patterns à Éviter

Anti-pattern	Problème	Solution
Pas de compaction	Dégradation performance	Automatiser la compaction
Compaction trop fréquente	Gaspillage de ressources	Seuils adaptés au volume
Expiration agressive	Perte de Time Travel	Rétention suffisante
Pas de monitoring	Problèmes non détectés	Alertes proactives
Maintenance manuelle	Oublis, incohérences	Automatisation complète
Même config pour tout	Sous-optimisation	Config par type de table

Conclusion

La maintenance d'un Lakehouse Apache Iceberg en production constitue une discipline à part entière, distincte de la conception et du développement initial. Les tables Iceberg, par leur nature immutable et versionnée, accumulent naturellement fichiers et métadonnées qui, sans gestion active, dégradent progressivement les performances et gonflent les coûts. La différence entre un Lakehouse performant et un système problématique réside souvent dans la qualité des pratiques opérationnelles plutôt que dans l'architecture initiale.

Les opérations fondamentales — compaction, expiration des snapshots, nettoyage des orphelins — doivent être orchestrées de manière automatisée et adaptée au profil de chaque table. Une table de streaming haute fréquence nécessite une compaction agressive toutes les heures, tandis qu'une table d'archive peut se contenter d'une optimisation mensuelle. Cette différenciation, guidée par les métriques et les patterns d'utilisation réels, optimise l'équilibre entre performance et coût de maintenance.

Le monitoring proactif transforme la maintenance de réactive à préventive. Des métriques bien choisies — ratio de petits fichiers, nombre de snapshots, fragmentation des manifestes — permettent d'anticiper les problèmes avant qu'ils n'impactent les utilisateurs. Les alertes automatisées et les dashboards de santé offrent une visibilité continue sur l'état du Lakehouse.

Les études de cas canadiennes démontrent que des organisations de tailles et secteurs variés ont réussi à établir des pratiques de maintenance robustes. Que ce soit une banque d'investissement avec 500 To de données réglementées, un opérateur télécoms gérant 50 milliards d'événements quotidiens, ou une chaîne d'épiceries avec une équipe réduite, des stratégies adaptées permettent de maintenir performance et fiabilité dans la durée.

Le chapitre suivant aborde l'opérationnalisation complète du Lakehouse, où nous examinerons les aspects organisationnels, les modèles d'exploitation et les pratiques DevOps/DataOps qui complètent les opérations techniques de maintenance.

Résumé

Opérations essentielles :

- **Compaction** : Consolide les petits fichiers, améliore les performances de lecture
- **Expiration snapshots** : Libère l'espace, contrôle la croissance des métadonnées
- **Nettoyage orphelins** : Supprime les fichiers déréférencés
- **Réécriture manifestes** : Optimise la structure des métadonnées

Paramètres clés de compaction :

- Taille cible : 256-512 Mo pour tables actives, 512 Mo-1 Go pour archives
- Seuil de déclenchement : 20-30% de petits fichiers
- Stratégie : binpack pour simple consolidation, sort/Z-order pour optimisation requêtes

Monitoring recommandé :

- Ratio petits fichiers (seuil: < 20%)
- Nombre de snapshots (seuil: < 100)
- Taille moyenne des fichiers (seuil: > 100 Mo)
- Nombre de manifestes (seuil: < 100)

Automatisation :

- Orchestration Airflow pour planification fiable
- Triggers événementiels pour réaction aux seuils
- Scripts réutilisables et paramétrables
- Alertes proactives vers Slack/courriel

Récupération :

- Rollback via snapshots pour erreurs d'écriture
- Sauvegarde régulière des métadonnées
- Réplication cross-région pour DR
- Procédures testées et documentées

Fréquences recommandées :

- Compaction : horaire à quotidienne selon volume
- Expiration : quotidienne
- Orphelins : hebdomadaire
- Manifestes : mensuelle

Ce chapitre établit les pratiques opérationnelles de votre Lakehouse. Le chapitre suivant, « Opérationnalisation et DevOps », explore les aspects organisationnels et les pratiques d'équipe pour une exploitation mature de la plateforme.

Chapitre IV.11 - OPÉRATIONNALISER APACHE ICEBERG

De la conception à l'exploitation : transformer votre lakehouse en plateforme de production

Introduction

Le passage d'un environnement de développement à une plateforme de production représente l'une des transitions les plus critiques dans le cycle de vie d'un Data Lakehouse. Concevoir une architecture Apache Iceberg élégante constitue un accomplissement technique, mais l'opérationnaliser de manière fiable, observable et résiliente représente un défi d'une tout autre nature. Cette distinction entre « ça fonctionne » et « ça fonctionne en production » sépare les projets expérimentaux des plateformes de données d'entreprise.

Les organisations qui adoptent Apache Iceberg découvrent rapidement que la valeur du format de table ne se réalise pleinement qu'avec une discipline opérationnelle rigoureuse. Sans orchestration appropriée, les pipelines s'exécutent de manière chaotique, les dépendances se brisent silencieusement et les opérations de maintenance s'accumulent. Sans audit, les exigences réglementaires deviennent ingérables et le diagnostic des problèmes tourne au cauchemar. Sans stratégie de récupération, un incident mineur peut se transformer en catastrophe.

Dans le chapitre précédent, nous avons exploré les mécanismes de maintenance d'un lakehouse Iceberg en production, notamment la compaction, l'expiration des snapshots et l'exploration des tables de métadonnées. Ce chapitre prolonge cette réflexion en abordant les dimensions opérationnelles plus larges : l'orchestration des pipelines, l'audit et la traçabilité des opérations, ainsi que la récupération après sinistre.

L'opérationnalisation d'Apache Iceberg s'inscrit dans la discipline émergente du DataOps, qui applique les principes DevOps au domaine des données. Cette approche reconnaît que les pipelines de données partagent les mêmes exigences que les applications logicielles en termes de fiabilité, d'observabilité et de récupération. Toutefois, les données présentent des caractéristiques uniques — leur volume, leur état et leur historique — qui nécessitent des stratégies adaptées.

Le contexte réglementaire canadien ajoute une couche de complexité supplémentaire. La Loi 25 au Québec, entrée pleinement en vigueur en 2024, impose des obligations strictes en matière de traçabilité et de protection des données personnelles. Les organisations assujetties doivent pouvoir démontrer où sont stockées les données, qui y accède, et comment elles sont transformées. Un lakehouse bien opérationnalisé, avec ses capacités d'audit et de *time travel*, répond naturellement à ces exigences lorsqu'il est correctement configuré.

Ce chapitre s'adresse aux architectes data et aux ingénieurs plateforme responsables de l'exploitation quotidienne d'un lakehouse Iceberg. Nous examinerons les outils d'orchestration modernes, les pratiques d'audit conformes aux exigences réglementaires et les stratégies de récupération après sinistre qui exploitent les capacités natives d'Iceberg. L'objectif est de fournir un cadre pratique permettant de transformer une architecture de données en une plateforme opérationnelle robuste.

IV.11.1 Orchestration du Lakehouse

L'orchestration constitue le système nerveux central de tout lakehouse en production. Elle coordonne l'exécution des pipelines de données, gère les dépendances entre tâches et assure la résilience face aux défaillances. Dans le contexte d'Apache Iceberg, l'orchestration doit également gérer les opérations de maintenance spécifiques au format de table, telles que la compaction et l'expiration des snapshots.

Le rôle de l'orchestration dans un lakehouse moderne

Un lakehouse Iceberg en production comprend typiquement des dizaines, voire des centaines de pipelines interconnectés. Ces pipelines ingèrent des données depuis diverses sources, appliquent des transformations, exécutent des opérations de maintenance et alimentent les consommateurs en aval. Sans orchestration centralisée, la coordination de ces processus devient rapidement ingérable.

L'orchestration remplit plusieurs fonctions essentielles dans ce contexte. Premièrement, elle assure la planification temporelle des tâches selon des calendriers définis ou des événements déclencheurs. Deuxièmement, elle gère les dépendances entre pipelines, garantissant qu'une transformation ne s'exécute qu'après l'ingestion réussie des données sources. Troisièmement, elle orchestre les opérations de reprise automatique en cas d'échec, avec des stratégies de backoff exponentielles pour éviter la surcharge des systèmes. Quatrièmement, elle fournit une vue unifiée de l'état de tous les pipelines, facilitant le diagnostic des problèmes.

L'orchestration moderne va au-delà de la simple planification de tâches. Elle intègre des concepts comme l'orchestration orientée données (*data-aware orchestration*), où les décisions d'exécution sont basées sur l'état des données plutôt que sur des horaires fixes. Cette approche s'aligne naturellement avec les capacités d'Iceberg, où les snapshots et les métadonnées fournissent une visibilité complète sur l'état des tables.

Panorama des orchestrateurs pour les lakehouses

Le paysage des outils d'orchestration a considérablement évolué ces dernières années. Selon l'analyse de l'écosystème des orchestrateurs en 2025 (PracData, 2025), Apache Airflow demeure le standard dominant avec plus de 320 millions de téléchargements annuels, suivi de Prefect et Dagster. Chaque outil présente des forces distinctes pour l'orchestration d'un lakehouse Iceberg.

Apache Airflow représente le choix éprouvé pour les organisations recherchant maturité et écosystème étendu. Son modèle basé sur les DAG (Directed Acyclic Graphs) définit les flux de travail comme du code Python, offrant flexibilité et versionnement. Airflow excelle dans les environnements où les pipelines sont relativement stables et où l'intégration avec des systèmes existants est primordiale. Ses milliers d'opérateurs prêts à l'emploi facilitent l'intégration avec Apache Spark, Trino et les services infonuagiques. Les versions gérées comme Amazon MWAA, Google Cloud Composer et Astronomer simplifient l'exploitation en production.

Toutefois, Airflow présente des limitations pour les lakehouses modernes. Son modèle centré sur les tâches plutôt que sur les données complique la gestion de la lignée et l'observabilité des actifs de données. Les performances peuvent se dégrader avec des déploiements à très grande échelle, nécessitant une optimisation attentive de l'ordonnanceur.

Dagster adopte une approche fondamentalement différente, centrée sur les actifs de données (*software-defined assets*). Plutôt que de définir des tâches, les ingénieurs définissent les données qu'ils souhaitent

produire, et Dagster gère automatiquement les dépendances et l'exécution. Cette philosophie s'aligne remarquablement bien avec les lakehouses, où les tables Iceberg constituent les actifs principaux.

Dagster offre plusieurs avantages pour l'orchestration Iceberg. Son système de partitionnement natif gère élégamment les tables partitionnées temporellement, courantes dans les lakehouses. Les *sensors* permettent de déclencher des pipelines en réponse à l'arrivée de nouvelles données, plutôt que selon des horaires fixes. Les vérifications de qualité des données (*asset checks*) s'intègrent directement dans la définition des actifs, facilitant la validation avant publication.

L'article de Dagster (2025) démontre comment construire un lakehouse complet avec MinIO, Trino et Iceberg en utilisant une approche orientée actifs. Cette architecture permet une observabilité native où chaque table Iceberg est visible avec son historique de matérialisation, ses dépendances et ses métriques de santé.

Prefect se positionne comme une alternative pythonique avec un modèle d'exécution hybride. La version 3.0 (2025) embrasse l'orchestration sans serveur (*serverless*), où les ressources de calcul sont provisionnées à la demande. Cette approche peut réduire significativement les coûts pour les pipelines à exécution intermittente.

Temporal mérite également une mention pour les cas d'usage impliquant des processus de longue durée ou des agents d'intelligence artificielle. Selon Datum Labs (2025), Temporal excelle dans la gestion d'états complexes et la récupération après des défaillances prolongées, ce qui peut être pertinent pour les pipelines d'apprentissage automatique alimentés par un lakehouse.

Comparaison des orchestrateurs pour les lakehouses

Le choix d'un orchestrateur dépend de multiples facteurs : maturité de l'équipe, complexité des pipelines, exigences de latence et intégration avec l'écosystème existant. Le tableau suivant synthétise les caractéristiques clés des principaux orchestrateurs pour les lakehouses Iceberg.

Critère	Apache Airflow	Dagster	Prefect	Temporal
Philosophie	Orienté tâches (DAG)	Orienté actifs (Assets)	Flux de données	Workflows durables
Maturité	Très mature (2014)	Mature (2019)	Mature (2018)	Mature (2019)
Communauté	Très large	Large	Moyenne	Moyenne
Courbe d'apprentissage	Moyenne	Moyenne-haute	Faible	Haute
Observabilité native	Moyenne	Excellente	Bonne	Bonne
Lignée des données	Limitée	Native	Limitée	N/A
Gestion des partitions	Manuelle	Native	Manuelle	N/A
Tests locaux	Limités	Excellents	Bons	Bons
Intégration infonuagique	Excellente (MWAA, Composer)	Dagster Cloud	Prefect Cloud	Temporal Cloud
Coût (auto-hébergé)	Moyen	Moyen	Faible	Moyen
Cas d'usage optimal	Pipelines batch stables	Lakehouses, ML	Pipelines agiles	Workflows IA

Critères de sélection d'un orchestrateur

La sélection d'un orchestrateur pour un lakehouse Iceberg doit considérer plusieurs dimensions.

Maturité de l'équipe data : Les équipes familières avec Airflow peuvent bénéficier de sa large communauté et de ses ressources de formation abondantes. Les équipes plus modernes ou construisant un nouveau lakehouse peuvent préférer Dagster pour son alignement avec les architectures orientées actifs.

Exigences de lignée : Si la traçabilité des données est critique (conformité réglementaire, debug de pipelines complexes), Dagster offre une lignée native supérieure. Airflow nécessite des outils complémentaires comme Apache Atlas ou des solutions commerciales.

Complexité des pipelines : Les pipelines simples avec peu de dépendances fonctionnent bien avec n'importe quel orchestrateur. Les architectures complexes avec des centaines de tables interconnectées bénéficient de l'approche orientée actifs de Dagster.

Budget et ressources : Airflow en mode auto-hébergé peut devenir coûteux en ressources d'administration. Les versions gérées (MWAA, Composer) simplifient l'exploitation mais ajoutent des coûts directs. Dagster Cloud et Prefect Cloud offrent des modèles de tarification basés sur l'usage.

Intégration avec l'IA/ML : Pour les organisations intégrant fortement l'apprentissage automatique, Dagster offre des intégrations natives avec MLflow, Weights & Biases et les frameworks de feature store. Temporal convient aux workflows d'agents IA nécessitant une gestion d'état durable.

Étude de cas : Recommandation d'orchestrateur *Contexte* : Une entreprise de commerce électronique canadienne avec 50 tables Iceberg, 20 data engineers et des exigences de conformité Loi 25. *Analyse* : L'équipe utilise actuellement des scripts cron et souhaite professionnaliser l'orchestration. La lignée des données est importante pour la conformité. Le budget est limité. *Recommandation* : Dagster en mode auto-hébergé sur Kubernetes. La lignée native répond aux exigences de conformité. L'approche orientée actifs s'aligne avec l'architecture Iceberg. Les coûts restent maîtrisés en évitant les solutions SaaS. *Alternative* : Airflow sur Amazon MWAA si l'équipe est déjà familière avec Airflow et préfère une exploitation simplifiée.

Modèles d'orchestration pour Apache Iceberg

L'orchestration d'un lakehouse Iceberg requiert des modèles spécifiques qui tiennent compte des caractéristiques du format de table. Nous identifions trois modèles principaux : l'orchestration par horaire, l'orchestration événementielle et l'orchestration hybride.

L'orchestration par horaire demeure le modèle le plus répandu, où les pipelines s'exécutent selon des calendriers prédéfinis. Un pipeline d'ingestion quotidien pourrait s'exécuter à minuit, suivi des transformations à 2h00 et des tâches de maintenance à 4h00. Ce modèle offre prévisibilité et simplicité, mais peut entraîner des latences importantes et une utilisation inefficace des ressources.

```
# Exemple Airflow : Pipeline Iceberg quotidien
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime, timedelta

default_args = {
    'owner': 'data-engineering',
    'depends_on_past': False,
    'retries': 3,
    'retry_delay': timedelta(minutes=5),
}

with DAG(
    'lakehouse_daily_pipeline',
    default_args=default_args,
    schedule_interval='0 0 * * *',
    start_date=datetime(2025, 1, 1),
    catchup=False
) as dag:

    ingest_task = PythonOperator(
        task_id='ingest_raw_data',
        python_callable=ingest_to_iceberg,
    )

    transform_task = PythonOperator(
        task_id='transform_silver_layer',
        python_callable=transform_silver,
    )

    compact_task = PythonOperator(
        task_id='compact_iceberg_tables',
        python_callable=run_compaction,
    )
```

```
    expire_snapshots_task = PythonOperator(
        task_id='expire_old_snapshots',
        python_callable=expire_snapshots,
    )

    ingest_task >> transform_task >> compact_task >> expire_snapshots_task
```

L'orchestration événementielle déclenche les pipelines en réponse à des événements, tels que l'arrivée de nouveaux fichiers dans le stockage objet ou la publication d'un message Kafka. Ce modèle réduit la latence et optimise l'utilisation des ressources, mais augmente la complexité opérationnelle.

L'intégration avec Apache Kafka, détaillée dans le Volume III de cette monographie, permet de créer des architectures de *Streaming Lakehouse* où les données s'écoulent continuellement vers les tables Iceberg. L'orchestrateur surveille alors les commits Iceberg ou les offsets Kafka pour déclencher les transformations en aval.

```
# Exemple Dagster : Sensor surveillant MinIO pour nouveaux fichiers
from dagster import sensor, RunRequest, SensorEvaluationContext
from datetime import datetime, timedelta

@sensor(
    minimum_interval_seconds=60,
    job_name="incremental_ingestion_job"
)
def minio_new_files_sensor(context: SensorEvaluationContext):
    """Surveille MinIO pour détecter les nouveaux fichiers et déclencher l'ingestion."""
    minio_client = context.resources.minio

    # Vérifier les fichiers des 24 dernières heures
    for days_back in range(1, 8):
        date = datetime.now() - timedelta(days=days_back)
        date_str = date.strftime("%Y-%m-%d")
        prefix = f"raw/transactions/dt={date_str}/"

        new_files = minio_client.list_new_objects(prefix)

        if new_files:
            yield RunRequest(
                partition_key=date_str,
                tags={"trigger": "new_data_detected", "source": "minio_sensor"}
            )
```

L'orchestration hybride combine les deux approches, utilisant des horaires pour les traitements batch réguliers et des événements pour les besoins de faible latence. Ce modèle représente l'état de l'art pour les lakehouses d'entreprise qui doivent supporter à la fois des rapports quotidiens et des tableaux de bord quasi temps réel.

Orchestrer les opérations de maintenance Iceberg

Les opérations de maintenance constituent un aspect critique de l'orchestration d'un lakehouse. La compaction, l'expiration des snapshots et le nettoyage des fichiers orphelins doivent être exécutés régulièrement pour maintenir des performances optimales.

L'orchestration de la compaction requiert une attention particulière. Exécuter la compaction trop fréquemment consomme des ressources inutilement ; l'exécuter trop rarement dégrade les performances de lecture. Une stratégie efficace consiste à surveiller les métriques des tables (nombre de fichiers, taille moyenne des fichiers) et à déclencher la compaction lorsque des seuils sont dépassés.

```
# Fonction de compaction conditionnelle
def conditional_compaction(table_name: str, spark_session):
    """Exécute la compaction uniquement si nécessaire."""
    from pyspark.sql.functions import count, avg

    # Récupérer les métriques des fichiers
    files_df = spark_session.read.format("iceberg").load(f"{table_name}.files")

    metrics = files_df.agg(
        count("*").alias("file_count"),
        avg("file_size_in_bytes").alias("avg_file_size")
    ).collect()[0]

    file_count = metrics["file_count"]
    avg_file_size = metrics["avg_file_size"]

    # Seuils de compaction
    MAX_FILES = 1000
    MIN_AVG_SIZE = 64 * 1024 * 1024  # 64 MB

    if file_count > MAX_FILES or avg_file_size < MIN_AVG_SIZE:
        spark_session.sql(f"""
            CALL iceberg.system.rewrite_data_files(
                table => '{table_name}',
                options => map('target-file-size-bytes', '134217728')
            )
        """)
        return {"compacted": True, "file_count": file_count}

    return {"compacted": False, "file_count": file_count}
```

L'expiration des snapshots doit être coordonnée avec les politiques de rétention de l'organisation et les exigences de *time travel*. Une pratique courante consiste à conserver les snapshots des 7 derniers jours pour permettre les analyses ad hoc, tout en maintenant des snapshots mensuels pour les audits.

```
-- Expiration des snapshots avec rétention de 7 jours
CALL iceberg.system.expire_snapshots(
    table => 'lakehouse.gold.customer_360',
    older_than => TIMESTAMP '2025-01-09 00:00:00',
    retain_last => 10
);

-- Nettoyage des fichiers orphelins (exécuter mensuellement)
CALL iceberg.system.remove_orphan_files(
    table => 'lakehouse.gold.customer_360',
    older_than => TIMESTAMP '2025-01-01 00:00:00'
);
```

Infrastructure as Code pour l'orchestration

L'approche *Infrastructure as Code* (IaC) s'applique naturellement à l'orchestration d'un lakehouse. Les définitions de pipelines, les configurations des tâches et les paramètres d'exécution doivent être versionnés dans un système de contrôle de version comme Git.

Cette approche offre plusieurs avantages. La traçabilité permet de comprendre l'évolution des pipelines et d'identifier les changements responsables de régressions. La reproductibilité garantit que le même code produit le même comportement dans tous les environnements. La collaboration facilite la revue par les pairs et l'amélioration continue des pipelines.

Les outils modernes comme Dagster et Prefect sont conçus pour cette approche, avec des définitions de pipelines en Python pur qui s'intègrent naturellement avec les pratiques de développement logiciel.

```
# Structure de projet recommandée pour un lakehouse orchestré
# lakehouse-orchestration/
# ├── dags/                    # Définitions Airflow
# │   ├── ingestion/
# │   ├── transformation/
# │   └── maintenance/
# ├── assets/                  # Définitions Dagster
# │   ├── bronze/
# │   ├── silver/
# │   └── gold/
# ├── configs/
# │   ├── dev.yaml
# │   ├── staging.yaml
# │   └── prod.yaml
# ├── tests/
# │   ├── unit/
# │   └── integration/
# └── requirements.txt
```

Le déploiement des pipelines doit suivre un flux CI/CD (Intégration Continue / Déploiement Continu) avec des tests automatisés. Les modifications passent d'abord par un environnement de développement, puis de staging, avant d'être promues en production.

```yaml
# Exemple GitHub Actions pour déploiement Airflow
name: Deploy DAGs

on:
  push:
    branches: [main]
    paths: ['dags/**']

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Test DAGs
        run: |
          pip install apache-airflow pytest
          pytest tests/unit/

  deploy:
    needs: test
```

```
    runs-on: ubuntu-latest
    steps:
      - name: Sync DAGs to S3
        run: |
          aws s3 sync dags/ s3://airflow-dags-bucket/dags/
```

Gestion des environnements multiples

Un lakehouse d'entreprise comprend typiquement plusieurs environnements : développement, test, staging et production. L'orchestration doit gérer ces environnements de manière cohérente tout en permettant des différences de configuration.

Les pratiques recommandées incluent la paramétrisation des pipelines pour externaliser les configurations spécifiques à l'environnement (chemins de stockage, tailles de cluster, politiques de rétention). La configuration par environnement utilise des fichiers YAML ou des variables d'environnement pour injecter les valeurs appropriées. L'isolation des données garantit que les environnements de développement utilisent des sous-ensembles ou des données anonymisées.

```
# Configuration multi-environnement avec Dagster
from dagster import ConfigurableResource
from pydantic import Field

class IcebergConfig(ConfigurableResource):
    """Configuration paramétrable par environnement."""
    catalog_uri: str = Field(description="URI du catalogue Iceberg")
    warehouse_path: str = Field(description="Chemin du warehouse")
    default_database: str = Field(description="Base de données par défaut")
    retention_days: int = Field(default=7, description="Rétention des snapshots")
    compaction_threshold_files: int = Field(default=500, description="Seuil de
compaction")

# Configurations par environnement
configs = {
    "dev": IcebergConfig(
        catalog_uri="http://nessie-dev:19120/api/v1",
        warehouse_path="s3a://lakehouse-dev/warehouse",
        default_database="dev_lakehouse",
        retention_days=3,
        compaction_threshold_files=100
    ),
    "prod": IcebergConfig(
        catalog_uri="http://nessie-prod:19120/api/v1",
        warehouse_path="s3a://lakehouse-prod/warehouse",
        default_database="prod_lakehouse",
        retention_days=30,
        compaction_threshold_files=500
    )
}
```

Bonnes pratiques d'orchestration

L'expérience opérationnelle révèle plusieurs bonnes pratiques pour l'orchestration d'un lakehouse Iceberg.

Idempotence des pipelines : Chaque tâche doit être idempotente, c'est-à-dire que son exécution répétée avec les mêmes entrées produit le même résultat. Cette propriété est essentielle pour les reprises automatiques. Iceberg facilite l'idempotence grâce aux opérations atomiques et au *time travel*, permettant de rejouer des transformations à partir d'un snapshot spécifique.

Observabilité native : L'orchestrateur doit émettre des métriques détaillées sur l'exécution des pipelines : durée, volume de données traitées, nombre de fichiers créés. Ces métriques alimentent les tableaux de bord opérationnels et les alertes.

Gestion des échecs : Les stratégies de reprise doivent être adaptées au type de défaillance. Une erreur de connexion réseau justifie une reprise immédiate avec backoff exponentiel ; une erreur de données nécessite une intervention humaine. Les alertes doivent distinguer ces cas pour éviter la fatigue des équipes.

Tests des pipelines : Les DAG et les définitions d'actifs doivent être testés comme du code applicatif. Dagster offre des facilités de test local particulièrement développées, permettant de valider les pipelines avant déploiement en production.

> **Étude de cas : Financière Sun Life** *Secteur* : Services financiers *Défi* : Orchestrer 150+ pipelines de données alimentant les analyses actuarielles et la conformité réglementaire, avec des fenêtres de traitement nocturnes de 4 heures maximum. *Solution* : Migration d'Airflow vers Dagster pour bénéficier de l'orchestration orientée actifs. Chaque table Iceberg (bronze, silver, gold) est définie comme un actif avec ses dépendances explicites. Les sensors surveillent les commits Iceberg pour déclencher les transformations en cascade. *Résultats* : Réduction de 40 % du temps de traitement global grâce à une meilleure parallélisation. Amélioration de la visibilité avec lignée automatique des données. Temps moyen de résolution des incidents réduit de 2 heures à 25 minutes.

IV.11.2 Audit du Lakehouse

L'audit constitue une dimension incontournable de l'opérationnalisation d'un lakehouse, particulièrement dans les secteurs réglementés comme la finance, la santé et les télécommunications. Au-delà des exigences de conformité, l'audit fournit une traçabilité essentielle pour le diagnostic des problèmes et l'analyse forensique des données.

Exigences d'audit dans un contexte réglementaire

Les réglementations modernes imposent des exigences strictes en matière de traçabilité des données. Le Règlement général sur la protection des données (RGPD) en Europe et la Loi 25 au Québec exigent que les organisations puissent démontrer comment les données personnelles sont collectées, transformées et utilisées. Les normes sectorielles comme SOX (secteur financier) ou HIPAA (santé) ajoutent des contraintes supplémentaires.

Ces exigences se traduisent en capacités d'audit spécifiques pour un lakehouse. La traçabilité des accès enregistre qui a accédé à quelles données, quand et depuis quelle application. La traçabilité des modifications documente chaque changement apporté aux données, incluant l'auteur, le moment et la nature de la modification. La lignée des données trace le parcours des données depuis leur source jusqu'à leur consommation, permettant de comprendre comment un rapport est produit à partir des données brutes.

Apache Iceberg offre des fondations solides pour répondre à ces exigences grâce à son architecture basée sur les snapshots. Chaque modification d'une table Iceberg crée un nouveau snapshot, préservant l'historique complet des changements. Cette immutabilité native constitue un avantage significatif par rapport aux systèmes traditionnels où les données sont modifiées en place.

Architecture d'audit pour un lakehouse Iceberg

Une architecture d'audit complète pour un lakehouse Iceberg comprend plusieurs composants : les journaux d'accès, les journaux de modification, la lignée des données et les rapports de conformité.

Journaux d'accès : Les moteurs de requête comme Dremio et Trino génèrent des journaux détaillant chaque requête exécutée. Ces journaux capturent l'identité de l'utilisateur, la requête SQL, les tables accédées, les données retournées (ou un résumé) et les performances. Ces journaux doivent être centralisés dans un système d'observabilité comme Elasticsearch/Kibana (ELK) ou une solution spécialisée.

```
-- Exemple de structure pour une table de journalisation des accès
CREATE TABLE audit.query_logs (
    query_id STRING,
    user_id STRING,
    user_email STRING,
    client_ip STRING,
    query_text STRING,
    tables_accessed ARRAY<STRING>,
    columns_accessed ARRAY<STRING>,
    rows_returned BIGINT,
    execution_time_ms BIGINT,
    query_status STRING,
    error_message STRING,
    timestamp TIMESTAMP
) USING iceberg
PARTITIONED BY (days(timestamp));
```

Journaux de modification : Iceberg maintient nativement l'historique des modifications à travers les snapshots. La table de métadonnées `snapshots` fournit un journal de toutes les opérations effectuées sur une table.

```
-- Consulter l'historique des modifications d'une table
SELECT
    snapshot_id,
    committed_at,
    operation,
    summary['added-data-files'] as files_added,
    summary['deleted-data-files'] as files_deleted,
    summary['added-records'] as records_added,
    summary['deleted-records'] as records_deleted
FROM lakehouse.gold.customer_360.snapshots
ORDER BY committed_at DESC
LIMIT 100;
```

Pour enrichir ces journaux avec des informations contextuelles (utilisateur, application source, raison de la modification), une pratique efficace consiste à utiliser les propriétés de snapshot. Lors de chaque écriture, l'application peut définir des propriétés personnalisées qui seront préservées dans les métadonnées.

```
# Écriture avec métadonnées d'audit
df.writeTo("lakehouse.gold.customer_360") \
    .option("snapshot-property.audit.user", current_user) \
    .option("snapshot-property.audit.application", "etl-pipeline-v2") \
    .option("snapshot-property.audit.reason", "Daily refresh from CRM") \
    .option("snapshot-property.audit.ticket", "JIRA-12345") \
    .append()
```

Lignée des données : La lignée documente les relations entre les tables, permettant de tracer le parcours des données. Cette information est cruciale pour comprendre l'impact d'un changement en amont sur les rapports en aval, et pour répondre aux demandes de conformité concernant la provenance des données.

Les outils de catalogues comme Apache Polaris, Atlan ou Alation peuvent extraire automatiquement la lignée en analysant les requêtes SQL et les définitions de pipelines. Cette lignée peut être enrichie par les orchestrateurs orientés actifs comme Dagster, qui maintiennent une lignée native basée sur les définitions de dépendances.

```
# Exemple Dagster : Définition d'actifs avec lignée explicite
from dagster import asset

@asset(
    description="Table agrégée des ventes par région",
    metadata={
        "owner": "equipe-analytics",
        "data_quality_score": 0.95,
        "pii_classification": "none"
    }
)
def ventes_par_region(transactions_nettoyees, reference_regions):
    """Agrège les transactions par région géographique."""
    # La lignée est automatiquement déduite des dépendances
    return transactions_nettoyees.join(reference_regions).groupBy("region").agg(...)
```

Intégration de la qualité des données dans l'audit

La qualité des données fait partie intégrante de l'audit d'un lakehouse. Les vérifications de qualité documentent l'état des données à chaque étape du pipeline, fournissant une piste d'audit de la fiabilité des données.

Les cadres de qualité des données comme Great Expectations, Soda ou dbt tests permettent de définir des attentes sur les données et de les valider automatiquement. Ces validations génèrent des rapports qui alimentent la piste d'audit.

```
# Exemple Great Expectations pour validation avec audit
import great_expectations as gx

context = gx.get_context()

# Définir les attentes pour la table customer_360
suite = context.add_expectation_suite("customer_360_quality")

# Attentes de complétude
suite.add_expectation(
    gx.expectations.ExpectColumnValuesToNotBeNull(
```

```
        column="customer_id",
        meta={"audit_category": "completeness", "severity": "critical"}
    )
)

# Attentes de validité
suite.add_expectation(
    gx.expectations.ExpectColumnValuesToBeBetween(
        column="age",
        min_value=0,
        max_value=150,
        meta={"audit_category": "validity", "severity": "warning"}
    )
)

# Attentes de cohérence
suite.add_expectation(
    gx.expectations.ExpectColumnPairValuesAToBeGreaterThanB(
        column_A="total_purchases",
        column_B="total_returns",
        or_equal=True,
        meta={"audit_category": "consistency", "severity": "warning"}
    )
)

# Exécuter la validation et stocker les résultats pour audit
checkpoint = context.add_checkpoint(
    name="customer_360_daily_validation",
    validations=[{
        "batch_request": {"datasource_name": "iceberg_lakehouse", "data_asset_name":
"customer_360"},
        "expectation_suite_name": "customer_360_quality"
    }]
)

results = checkpoint.run()

# Les résultats sont automatiquement stockés pour audit
print(f"Validation réussie : {results.success}")
print(f"Statistiques : {results.statistics}")
```

Les résultats de validation doivent être persistés dans une table d'audit dédiée, permettant l'analyse historique de la qualité des données.

```
-- Table d'audit de qualité des données
CREATE TABLE audit.data_quality_results (
    validation_id STRING,
    table_name STRING,
    validation_time TIMESTAMP,
    expectation_type STRING,
    column_name STRING,
    success BOOLEAN,
    observed_value STRING,
    expected_value STRING,
    severity STRING,
    audit_category STRING,
    snapshot_id BIGINT
```

```
) USING iceberg
PARTITIONED BY (days(validation_time));
```

Architecture de collecte des journaux d'audit

Une architecture robuste de collecte des journaux d'audit doit gérer le volume, la fiabilité et la rétention des données d'audit.

Collecte des journaux : Les journaux proviennent de multiples sources : moteurs de requête (Dremio, Trino, Spark), orchestrateurs (Airflow, Dagster), applications métier et services infonuagiques. Un agent de collecte comme Fluentd ou Vector agrège ces journaux et les achemine vers un système centralisé.

```
# Configuration Vector pour collecte des journaux d'audit
sources:
  trino_logs:
    type: file
    include: ["/var/log/trino/query*.log"]

  spark_logs:
    type: file
    include: ["/var/log/spark/audit*.log"]

  airflow_logs:
    type: file
    include: ["/var/log/airflow/scheduler*.log"]

transforms:
  parse_trino:
    type: remap
    inputs: ["trino_logs"]
    source: |
      . = parse_json!(.message)
      .source = "trino"
      .audit_timestamp = now()

sinks:
  iceberg_audit:
    type: http
    inputs: ["parse_trino", "parse_spark", "parse_airflow"]
    uri: "http://spark-rest:8080/audit/ingest"
    encoding:
      codec: json
```

Rétention et archivage : Les journaux d'audit doivent être conservés selon les exigences réglementaires, souvent 7 ans pour les données financières. Une stratégie de tiering déplace les journaux anciens vers un stockage économique (S3 Glacier, Azure Archive) tout en maintenant leur accessibilité.

```
-- Politique de rétention pour les journaux d'audit
-- Tables actives (< 90 jours) : Stockage standard
-- Tables archivées (90 jours - 7 ans) : Stockage archive

-- Vue unifiée des journaux d'audit
CREATE VIEW audit.unified_query_logs AS
SELECT * FROM audit.query_logs_active
UNION ALL
```

```
SELECT * FROM audit.query_logs_archive
WHERE timestamp > CURRENT_DATE - INTERVAL 7 YEAR;
```

Exploiter les tables de métadonnées pour l'audit

Iceberg expose plusieurs tables de métadonnées qui constituent des sources d'information précieuses pour l'audit. Ces tables sont accessibles via des requêtes SQL standard, facilitant leur intégration dans les processus d'audit existants.

Table `history` : Fournit l'historique des snapshots avec les métadonnées associées.

```
-- Historique complet des modifications
SELECT
    made_current_at,
    snapshot_id,
    parent_id,
    is_current_ancestor
FROM lakehouse.gold.customer_360.history
ORDER BY made_current_at DESC;
```

Table `snapshots` : Détaille chaque snapshot avec les statistiques d'opération.

Table `manifests` : Liste les fichiers manifest pour chaque snapshot, permettant une analyse fine des modifications au niveau des fichiers.

Table `files` : Énumère tous les fichiers de données avec leurs statistiques (taille, nombre d'enregistrements, valeurs min/max par colonne).

```
-- Analyse des fichiers pour détecter des anomalies
SELECT
    partition,
    COUNT(*) as file_count,
    SUM(record_count) as total_records,
    AVG(file_size_in_bytes) / 1024 / 1024 as avg_file_size_mb,
    MIN(file_size_in_bytes) / 1024 as min_file_size_kb,
    MAX(file_size_in_bytes) / 1024 / 1024 as max_file_size_mb
FROM lakehouse.gold.customer_360.files
GROUP BY partition
HAVING MIN(file_size_in_bytes) < 1024 * 1024  -- Fichiers < 1 MB
ORDER BY file_count DESC;
```

Observabilité et alertes

L'observabilité transforme les données d'audit brutes en informations actionnables. Elle permet de détecter proactivement les anomalies, d'alerter les équipes et de faciliter le diagnostic des problèmes.

Métriques clés à surveiller :

Métrique	Description	Seuil d'alerte suggéré
Nombre de fichiers par partition	Indicateur de fragmentation	> 500 fichiers
Taille moyenne des fichiers	Performance de lecture	< 32 MB ou > 512 MB
Latence d'ingestion	Fraîcheur des données	> 15 minutes du SLA
Taux d'échec des requêtes	Santé du système	> 1 %
Durée de compaction	Efficacité de maintenance	> 2x moyenne historique
Croissance du stockage	Coûts et capacité	> 20 % /semaine

Architecture d'observabilité : Une architecture typique comprend la collecte des métriques via Prometheus ou DataDog, la visualisation via Grafana, et les alertes configurées pour notifier les équipes via Slack, PagerDuty ou courriel.

L'architecture d'observabilité pour un lakehouse Iceberg doit couvrir plusieurs dimensions : la santé des tables, la performance des requêtes, l'état des pipelines et les métriques d'infrastructure.

```
# Configuration Prometheus pour métriques Iceberg
global:
  scrape_interval: 30s
  evaluation_interval: 30s

rule_files:
  - /etc/prometheus/rules/iceberg_alerts.yml

scrape_configs:
  - job_name: 'spark-metrics'
    static_configs:
      - targets: ['spark-master:4040']
    metrics_path: /metrics/prometheus

  - job_name: 'trino-metrics'
    static_configs:
      - targets: ['trino-coordinator:8080']
    metrics_path: /v1/jmx

  - job_name: 'nessie-metrics'
    static_configs:
      - targets: ['nessie:19120']
    metrics_path: /api/v2/metrics
```

```
# Règles d'alerte Prometheus pour le lakehouse
groups:
  - name: iceberg_table_health
    rules:
      - alert: IcebergTableFragmented
        expr: iceberg_table_file_count > 500
        for: 30m
```

```
        labels:
          severity: warning
          team: data-engineering
        annotations:
          summary: "Table Iceberg fragmentée: {{ $labels.table_name }}"
          description: "La table {{ $labels.table_name }} contient {{ $value }} fichiers.
Compaction recommandée."
          runbook_url: "https://runbooks.example.com/iceberg/compaction"

      - alert: IcebergIngestionLatency
        expr: iceberg_last_commit_age_seconds > 3600
        for: 15m
        labels:
          severity: critical
          team: data-engineering
        annotations:
          summary: "Retard d'ingestion: {{ $labels.table_name }}"
          description: "La table {{ $labels.table_name }} n'a pas été mise à jour depuis
{{ $value | humanizeDuration }}."

      - alert: IcebergSnapshotGrowth
        expr: rate(iceberg_snapshot_count[1h]) > 100
        for: 1h
        labels:
          severity: warning
        annotations:
          summary: "Croissance rapide des snapshots: {{ $labels.table_name }}"
          description: "La table génère des snapshots à un rythme élevé. Vérifier la
fréquence d'écriture."

      - alert: IcebergStorageGrowth
        expr: rate(iceberg_table_size_bytes[24h]) / iceberg_table_size_bytes > 0.2
        for: 2h
        labels:
          severity: warning
        annotations:
          summary: "Croissance du stockage > 20%/jour: {{ $labels.table_name }}"
          description: "Croissance inhabituelle du stockage. Vérifier les politiques de
rétention."

  - name: iceberg_query_performance
    rules:
      - alert: IcebergSlowQueries
        expr: histogram_quantile(0.95,
rate(query_execution_time_seconds_bucket{engine="trino"}[5m])) > 30
        for: 10m
        labels:
          severity: warning
        annotations:
          summary: "Requêtes lentes détectées"
          description: "Le 95e percentile des requêtes dépasse 30 secondes."

      - alert: IcebergQueryFailureRate
        expr: rate(query_failures_total[5m]) / rate(query_total[5m]) > 0.01
        for: 5m
        labels:
          severity: critical
        annotations:
```

```
          summary: "Taux d'échec des requêtes élevé"
          description: "Plus de 1% des requêtes échouent. Investiguer immédiatement."
```

Intégration avec les plateformes d'observabilité

Les organisations utilisent souvent des plateformes d'observabilité intégrées qui combinent métriques, journaux et traces. L'intégration avec ces plateformes est essentielle pour une vision unifiée.

Intégration DataDog : DataDog offre des intégrations natives pour de nombreux composants du lakehouse, incluant Spark, Trino et AWS S3.

```
# Émission de métriques personnalisées vers DataDog
from datadog import initialize, statsd
from pyspark.sql import SparkSession

initialize(statsd_host='datadog-agent', statsd_port=8125)

def emit_table_metrics(spark: SparkSession, table_name: str):
    """Émet les métriques de santé d'une table Iceberg vers DataDog."""

    # Récupérer les statistiques de fichiers
    files_df = spark.sql(f"""
        SELECT
            COUNT(*) as file_count,
            AVG(file_size_in_bytes) as avg_file_size,
            SUM(file_size_in_bytes) as total_size,
            SUM(record_count) as total_records
        FROM {table_name}.files
    """).collect()[0]

    # Récupérer les statistiques de snapshots
    snapshots_df = spark.sql(f"""
        SELECT
            COUNT(*) as snapshot_count,
            MAX(committed_at) as last_commit
        FROM {table_name}.snapshots
    """).collect()[0]

    # Émettre les métriques
    tags = [f"table:{table_name}", "lakehouse:prod"]

    statsd.gauge('iceberg.table.file_count', files_df['file_count'], tags=tags)
    statsd.gauge('iceberg.table.avg_file_size_mb',
                 files_df['avg_file_size'] / 1024 / 1024, tags=tags)
    statsd.gauge('iceberg.table.total_size_gb',
                 files_df['total_size'] / 1024 / 1024 / 1024, tags=tags)
    statsd.gauge('iceberg.table.total_records', files_df['total_records'], tags=tags)
    statsd.gauge('iceberg.table.snapshot_count', snapshots_df['snapshot_count'],
tags=tags)

    # Calculer l'âge du dernier commit
    from datetime import datetime
    if snapshots_df['last_commit']:
        last_commit_age = (datetime.utcnow() -
snapshots_df['last_commit']).total_seconds()
        statsd.gauge('iceberg.table.last_commit_age_seconds', last_commit_age, tags=tags)
```

Intégration OpenTelemetry : OpenTelemetry fournit un standard ouvert pour la collecte de métriques, traces et journaux. Cette approche offre flexibilité et évite le verrouillage fournisseur.

```
# Configuration OpenTelemetry pour le lakehouse
from opentelemetry import trace, metrics
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.metrics import MeterProvider
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import OTLPSpanExporter
from opentelemetry.exporter.otlp.proto.grpc.metric_exporter import OTLPMetricExporter

# Configuration du traceur
trace.set_tracer_provider(TracerProvider())
tracer = trace.get_tracer("lakehouse.iceberg")

otlp_exporter = OTLPSpanExporter(endpoint="otel-collector:4317", insecure=True)
trace.get_tracer_provider().add_span_processor(
    BatchSpanProcessor(otlp_exporter)
)

# Configuration des métriques
metrics.set_meter_provider(MeterProvider())
meter = metrics.get_meter("lakehouse.iceberg")

# Création des instruments de métrique
file_count_gauge = meter.create_observable_gauge(
    "iceberg.table.file_count",
    callbacks=[lambda: get_file_counts()],
    description="Nombre de fichiers par table Iceberg"
)

query_duration_histogram = meter.create_histogram(
    "iceberg.query.duration",
    unit="ms",
    description="Durée des requêtes Iceberg"
)

# Exemple d'instrumentation d'une opération
def instrumented_write(df, table_name: str):
    """Écriture instrumentée avec tracing."""
    with tracer.start_as_current_span("iceberg.write") as span:
        span.set_attribute("table.name", table_name)
        span.set_attribute("records.count", df.count())

        start_time = time.time()
        df.writeTo(table_name).append()
        duration_ms = (time.time() - start_time) * 1000

        query_duration_histogram.record(duration_ms, {"operation": "write", "table":
table_name})
        span.set_attribute("duration.ms", duration_ms)
```

Les plateformes d'observabilité des données comme Monte Carlo, Bigeye ou Great Expectations complètent cette architecture en surveillant la qualité des données elles-mêmes : distribution des valeurs, valeurs nulles inattendues, anomalies de volume.

> **Performance** L'exploitation intensive des tables de métadonnées Iceberg pour l'audit peut impacter les performances du catalogue, particulièrement pour les tables avec un historique long. Une bonne pratique consiste à extraire périodiquement ces métadonnées vers des tables d'audit dédiées, réduisant la charge sur le catalogue principal.

Conformité et rapports réglementaires

Les organisations assujetties à des réglementations doivent produire régulièrement des rapports de conformité démontrant la traçabilité et la protection des données.

Rapports RGPD/Loi 25 : Ces rapports documentent le traitement des données personnelles, incluant les bases légales du traitement, les durées de rétention et les accès effectués.

```
-- Rapport d'accès aux données personnelles (derniers 90 jours)
SELECT
    q.user_email,
    t.table_name,
    COUNT(*) as access_count,
    MAX(q.timestamp) as last_access,
    t.pii_classification
FROM audit.query_logs q
JOIN metadata.table_classifications t
    ON ARRAY_CONTAINS(q.tables_accessed, t.table_name)
WHERE q.timestamp > CURRENT_TIMESTAMP - INTERVAL 90 DAY
    AND t.pii_classification IN ('PII', 'Sensitive-PII')
GROUP BY q.user_email, t.table_name, t.pii_classification
ORDER BY access_count DESC;
```

Rapports SOX : Pour les entreprises cotées en bourse, les rapports SOX documentent les contrôles sur les données financières, incluant la séparation des tâches et les approbations.

Demandes de droit d'accès : Le *time travel* d'Iceberg facilite la réponse aux demandes de droit d'accès des individus en permettant de reconstituer l'état des données personnelles à une date donnée.

```
-- Reconstituer l'état des données d'un client à une date spécifique
SELECT *
FROM lakehouse.gold.customer_360
FOR TIMESTAMP AS OF TIMESTAMP '2024-06-15 00:00:00'
WHERE customer_id = 'CUST-12345';
```

> **Étude de cas : Mouvement Desjardins** *Secteur* : Services financiers *Défi* : Répondre aux exigences de l'Autorité des marchés financiers (AMF) et de la Loi 25 pour la traçabilité des données des 7 millions de membres. *Solution* : Déploiement d'une architecture d'audit basée sur les tables de métadonnées Iceberg, enrichie par des journaux applicatifs centralisés dans Elasticsearch. Chaque modification de données membres est associée à un ticket de changement et un approbateur. *Résultats* : Temps de réponse aux demandes de l'AMF réduit de 5 jours à 4 heures. Capacité de produire un rapport complet de lignée pour n'importe quel point de données en moins de 30 minutes. Conformité Loi 25 attestée par audit externe.

IV.11.3 Récupération Après Sinistre

La récupération après sinistre (Disaster Recovery, DR) constitue l'ultime filet de sécurité d'un lakehouse en production. Malgré toutes les précautions, des événements catastrophiques peuvent survenir : défaillance d'une région infonuagique, corruption de données à grande échelle, erreur humaine dévastatrice ou cyberattaque. Une stratégie de DR robuste garantit la continuité des opérations face à ces scénarios.

Définir les objectifs de récupération

Avant de concevoir une stratégie de DR, l'organisation doit définir deux métriques fondamentales pour chaque classe de données.

RTO (Recovery Time Objective) : Le temps maximum acceptable entre la survenue d'un sinistre et la restauration du service. Un RTO de 4 heures signifie que le lakehouse doit être opérationnel dans les 4 heures suivant l'incident.

RPO (Recovery Point Objective) : La quantité maximale de données que l'organisation peut se permettre de perdre. Un RPO de 1 heure signifie que les sauvegardes doivent avoir moins d'une heure d'ancienneté.

Ces objectifs varient selon la criticité des données. Les tables alimentant les systèmes transactionnels peuvent exiger un RTO de 15 minutes et un RPO de 5 minutes, tandis que les données d'archivage peuvent tolérer un RTO de 24 heures et un RPO d'une semaine.

Classe de données	Exemple	RTO cible	RPO cible
Critique	Données clients actifs	15 min	5 min
Important	Rapports financiers	2 heures	1 heure
Standard	Analyses marketing	8 heures	4 heures
Archive	Données historiques	24 heures	1 semaine

Capacités natives d'Iceberg pour la récupération

L'architecture d'Apache Iceberg offre des capacités de récupération supérieures aux formats de données traditionnels, grâce à plusieurs caractéristiques fondamentales.

Immutabilité des fichiers : Les fichiers de données Iceberg ne sont jamais modifiés une fois écrits. Chaque modification crée de nouveaux fichiers, préservant l'historique complet. Cette immutabilité signifie qu'une corruption ou suppression accidentelle n'affecte pas les snapshots historiques tant que leurs fichiers sont préservés.

Snapshots comme points de récupération : Chaque snapshot représente un état cohérent et complet de la table. La récupération vers un snapshot antérieur est une opération atomique qui ne nécessite aucune restauration de fichiers.

Séparation métadonnées/données : Les métadonnées Iceberg (fichiers JSON et manifest) sont distinctes des fichiers de données. Cette séparation permet des stratégies de sauvegarde différenciées et facilite la récupération en cas de corruption partielle.

Procédures de récupération intégrées : Iceberg fournit des procédures Spark pour les opérations de récupération courantes, notamment `rollback_to_snapshot`, `rollback_to_timestamp` et `register_table`.

Scénarios de récupération et solutions

L'article de Dremio (2025) sur la récupération après sinistre pour les tables Iceberg identifie plusieurs scénarios de défaillance et leurs solutions.

Scénario 1 : Erreur humaine — suppression ou modification accidentelle de données

Ce scénario représente la cause la plus fréquente de perte de données. Un ingénieur exécute par erreur un `DELETE` sans clause `WHERE`, ou un script de migration corrompt des enregistrements.

Solution : Utiliser la procédure `rollback_to_snapshot` ou `rollback_to_timestamp` pour revenir à l'état antérieur à l'erreur.

```
-- Identifier le snapshot cible
SELECT snapshot_id, committed_at, operation, summary
FROM lakehouse.gold.customer_360.snapshots
WHERE committed_at < TIMESTAMP '2025-01-15 14:30:00'
ORDER BY committed_at DESC
LIMIT 5;

-- Revenir au snapshot avant l'erreur
CALL iceberg.system.rollback_to_snapshot(
    'lakehouse.gold.customer_360',
    8744736658442914487
);

-- Ou revenir à un horodatage spécifique
CALL iceberg.system.rollback_to_timestamp(
    'lakehouse.gold.customer_360',
    TIMESTAMP '2025-01-15 14:00:00'
);
```

Cette récupération est quasi instantanée car elle ne modifie que le pointeur du snapshot courant, sans copier de données.

Scénario 2 : Corruption du catalogue

Le catalogue Iceberg (Hive Metastore, Nessie, REST Catalog) peut être corrompu ou indisponible, rendant les tables inaccessibles même si les fichiers de données sont intacts.

Solution : Utiliser la procédure `register_table` pour recréer l'entrée du catalogue en pointant vers le fichier de métadonnées le plus récent.

```
-- Ré-enregistrer une table à partir de son fichier metadata.json
CALL spark_catalog.system.register_table(
    table => 'lakehouse.gold.customer_360',
    metadata_file => 's3a://lakehouse-bucket/gold/customer_360/metadata/v17.metadata.json'
);
```

> **Migration** *De* : Catalogue corrompu ou inaccessible *Vers* : Table réenregistrée dans un nouveau catalogue *Stratégie* : Localiser le fichier `metadata.json` le plus récent dans le répertoire `metadata/` de la table, puis utiliser `register_table` pour recréer l'entrée du catalogue. Maintenir un inventaire des chemins metadata.json comme partie de la documentation de DR.

Scénario 3 : Restauration depuis une sauvegarde vers un nouvel emplacement

Lors d'une restauration de sauvegarde, les fichiers peuvent être copiés vers un emplacement différent de l'original. Les métadonnées Iceberg utilisent des chemins absolus, qui deviennent alors invalides.

Solution : Utiliser la procédure `rewrite_table_path` pour mettre à jour tous les chemins dans les métadonnées.

```
-- Réécrire les chemins après restauration vers un nouvel emplacement
CALL spark_catalog.system.rewrite_table_path(
    table => 'lakehouse.gold.customer_360',
    source_prefix => 'hdfs://old-cluster/warehouse/customer_360',
    target_prefix => 's3a://dr-bucket/warehouse/customer_360'
);
```

Cette procédure génère un fichier CSV listant tous les fichiers à copier physiquement vers le nouvel emplacement.

Scénario 4 : Perte complète d'une région infonuagique

Ce scénario catastrophique nécessite une réplication préventive vers une région secondaire.

Solution : Implémenter une réplication continue des fichiers de données et de métadonnées vers une région DR. Les fournisseurs de stockage objet comme AWS S3 et Azure Blob offrent des fonctionnalités de réplication inter-région. Une approche complémentaire consiste à utiliser des outils comme Apache DistCp pour Hadoop ou des solutions de réplication spécialisées.

Stratégie de sauvegarde pour un lakehouse Iceberg

Une stratégie de sauvegarde complète pour un lakehouse Iceberg doit couvrir trois composants : les métadonnées du catalogue, les métadonnées des tables et les fichiers de données.

Sauvegarde des métadonnées du catalogue : Le catalogue (Hive Metastore, base de données Nessie, etc.) doit être sauvegardé régulièrement. Pour un Hive Metastore MySQL, cela signifie des sauvegardes de base de données. Pour Nessie, l'exportation du journal des commits.

Sauvegarde des métadonnées Iceberg : Les fichiers `metadata.json` et les manifests doivent être inclus dans les sauvegardes. Une bonne pratique consiste à maintenir un inventaire des fichiers metadata.json pour chaque table, avec leur horodatage.

```
# Script de catalogage des fichiers metadata.json pour DR
import boto3
from datetime import datetime

def catalog_metadata_files(bucket_name: str, table_prefix: str):
    """Catalogue les fichiers metadata.json pour faciliter la récupération."""
    s3 = boto3.client('s3')

    paginator = s3.get_paginator('list_objects_v2')
    pages = paginator.paginate(
        Bucket=bucket_name,
        Prefix=f"{table_prefix}/metadata/"
    )

    metadata_files = []
    for page in pages:
        for obj in page.get('Contents', []):
            if obj['Key'].endswith('.metadata.json'):
```

```
                metadata_files.append({
                    'path': f"s3a://{bucket_name}/{obj['Key']}",
                    'last_modified': obj['LastModified'].isoformat(),
                    'size_bytes': obj['Size']
                })

    # Trier par date, le plus récent en premier
    metadata_files.sort(key=lambda x: x['last_modified'], reverse=True)

    return metadata_files
```

Sauvegarde des fichiers de données : Pour les fichiers de données (Parquet, ORC), la stratégie dépend des objectifs RPO. Une réplication synchrone vers une région secondaire offre un RPO proche de zéro mais augmente les coûts. Une réplication asynchrone quotidienne offre un compromis coût/RPO acceptable pour de nombreux cas d'usage.

Configuration de la réplication S3 Cross-Region

AWS S3 offre une fonctionnalité de réplication inter-région (CRR) qui peut être configurée pour répliquer automatiquement les fichiers du lakehouse vers une région DR. Cette configuration est essentielle pour les organisations déployées sur AWS.

```
{
  "Role": "arn:aws:iam::123456789012:role/s3-replication-role",
  "Rules": [
    {
      "ID": "ReplicateLakehouseData",
      "Status": "Enabled",
      "Priority": 1,
      "Filter": {
        "Prefix": "warehouse/"
      },
      "Destination": {
        "Bucket": "arn:aws:s3:::lakehouse-dr-bucket",
        "StorageClass": "STANDARD",
        "ReplicationTime": {
          "Status": "Enabled",
          "Time": {
            "Minutes": 15
          }
        },
        "Metrics": {
          "Status": "Enabled",
          "EventThreshold": {
            "Minutes": 15
          }
        }
      },
      "DeleteMarkerReplication": {
        "Status": "Enabled"
      }
    }
  ]
}
```

Pour Azure Blob Storage, la fonctionnalité équivalente s'appelle *Object Replication*. Pour Google Cloud Storage, il s'agit de *Dual-region* ou *Multi-region* buckets combinés avec des politiques de réplication personnalisées.

Création d'un inventaire DR complet

Un inventaire DR complet facilite la récupération en documentant tous les composants du lakehouse et leurs emplacements de sauvegarde.

```
# Script complet d'inventaire DR
import boto3
import json
from datetime import datetime
from typing import List, Dict

class DRInventoryManager:
    """Gère l'inventaire de récupération après sinistre pour un lakehouse Iceberg."""

    def __init__(self, warehouse_bucket: str, inventory_bucket: str):
        self.warehouse_bucket = warehouse_bucket
        self.inventory_bucket = inventory_bucket
        self.s3 = boto3.client('s3')

    def list_all_tables(self) -> List[str]:
        """Découvre toutes les tables Iceberg dans le warehouse."""
        tables = []
        paginator = self.s3.get_paginator('list_objects_v2')

        # Rechercher les répertoires metadata/ qui indiquent une table Iceberg
        pages = paginator.paginate(
            Bucket=self.warehouse_bucket,
            Delimiter='/'
        )

        for page in pages:
            for prefix in page.get('CommonPrefixes', []):
                table_prefix = prefix['Prefix'].rstrip('/')
                # Vérifier si c'est une table Iceberg (présence de metadata/)
                try:
                    self.s3.head_object(
                        Bucket=self.warehouse_bucket,
                        Key=f"{table_prefix}/metadata/"
                    )
                    tables.append(table_prefix)
                except:
                    pass

        return tables

    def get_table_metadata_files(self, table_prefix: str) -> List[Dict]:
        """Liste tous les fichiers metadata.json pour une table."""
        metadata_files = []
        paginator = self.s3.get_paginator('list_objects_v2')

        pages = paginator.paginate(
            Bucket=self.warehouse_bucket,
            Prefix=f"{table_prefix}/metadata/"
```

```
        )

        for page in pages:
            for obj in page.get('Contents', []):
                if obj['Key'].endswith('.metadata.json'):
                    metadata_files.append({
                        'path': f"s3a://{self.warehouse_bucket}/{obj['Key']}",
                        'last_modified': obj['LastModified'].isoformat(),
                        'size_bytes': obj['Size'],
                        'version': self._extract_version(obj['Key'])
                    })

        metadata_files.sort(key=lambda x: x['last_modified'], reverse=True)
        return metadata_files

    def _extract_version(self, key: str) -> int:
        """Extrait le numéro de version du nom de fichier metadata."""
        import re
        match = re.search(r'v(\d+)\.metadata\.json$', key)
        return int(match.group(1)) if match else 0

    def create_full_inventory(self) -> Dict:
        """Crée un inventaire complet du lakehouse pour DR."""
        tables = self.list_all_tables()

        inventory = {
            'generated_at': datetime.utcnow().isoformat(),
            'warehouse_bucket': self.warehouse_bucket,
            'table_count': len(tables),
            'tables': []
        }

        for table_path in tables:
            metadata_files = self.get_table_metadata_files(table_path)

            if metadata_files:
                table_info = {
                    'table_path': table_path,
                    'full_name': table_path.replace('/', '.'),
                    'latest_metadata': metadata_files[0]['path'],
                    'latest_version': metadata_files[0]['version'],
                    'metadata_count': len(metadata_files),
                    'last_updated': metadata_files[0]['last_modified'],
                    'recovery_commands': self._generate_recovery_commands(
                        table_path,
                        metadata_files[0]['path']
                    )
                }
                inventory['tables'].append(table_info)

        return inventory

    def _generate_recovery_commands(self, table_path: str, metadata_file: str) -> Dict:
        """Génère les commandes de récupération pour une table."""
        table_name = table_path.replace('/', '.')
        return {
            'register_table': f"""
CALL spark_catalog.system.register_table(
    table => '{table_name}',
```

```
    metadata_file => '{metadata_file}'
);""",
            'verify_table': f"""
SELECT COUNT(*) as row_count,
       MAX(updated_at) as last_update
FROM {table_name};""",
            'list_snapshots': f"""
SELECT snapshot_id, committed_at, operation
FROM {table_name}.snapshots
ORDER BY committed_at DESC
LIMIT 10;"""
        }

    def save_inventory(self, inventory: Dict) -> str:
        """Sauvegarde l'inventaire dans le bucket de sauvegarde."""
        inventory_key = f"dr-inventory/{datetime.utcnow().strftime('%Y/%m/%d/%H%M%S')}/
inventory.json"

        self.s3.put_object(
            Bucket=self.inventory_bucket,
            Key=inventory_key,
            Body=json.dumps(inventory, indent=2, default=str),
            ContentType='application/json'
        )

        # Créer également un pointeur vers la dernière version
        self.s3.put_object(
            Bucket=self.inventory_bucket,
            Key="dr-inventory/latest.json",
            Body=json.dumps(inventory, indent=2, default=str),
            ContentType='application/json'
        )

        return f"s3://{self.inventory_bucket}/{inventory_key}"


# Utilisation
if __name__ == "__main__":
    manager = DRInventoryManager(
        warehouse_bucket="lakehouse-prod",
        inventory_bucket="lakehouse-dr-backups"
    )

    inventory = manager.create_full_inventory()
    location = manager.save_inventory(inventory)

    print(f"Inventaire DR créé : {location}")
    print(f"Tables documentées : {inventory['table_count']}")
```

Automatisation des tests de récupération

Les tests de récupération doivent être automatisés et exécutés régulièrement. Un pipeline de test DR valide la capacité à restaurer les tables critiques dans les délais RTO définis.

```
# Pipeline de test DR automatisé
from datetime import datetime, timedelta
```

```
import time
from typing import List, Dict, Optional

class DRTestRunner:
    """Exécute et valide les tests de récupération après sinistre."""

    def __init__(self, spark_session, dr_config: Dict):
        self.spark = spark_session
        self.config = dr_config
        self.results: List[Dict] = []

    def test_snapshot_rollback(self, table_name: str, max_rto_seconds: int) -> Dict:
        """Teste la récupération par rollback de snapshot."""
        start_time = time.time()

        try:
            # Identifier le snapshot cible (24h avant)
            target_time = datetime.utcnow() - timedelta(hours=24)

            snapshots = self.spark.sql(f"""
                SELECT snapshot_id, committed_at
                FROM {table_name}.snapshots
                WHERE committed_at < '{target_time.isoformat()}'
                ORDER BY committed_at DESC
                LIMIT 1
            """).collect()

            if not snapshots:
                return self._record_result(table_name, "rollback", False,
                    "Aucun snapshot disponible", time.time() - start_time)

            target_snapshot = snapshots[0]['snapshot_id']

            # Créer une branche de test pour ne pas affecter la production
            self.spark.sql(f"""
                ALTER TABLE {table_name}
                CREATE BRANCH dr_test_{datetime.utcnow().strftime('%Y%m%d%H%M%S')}
                AS OF VERSION {target_snapshot}
            """)

            branch_name = f"dr_test_{datetime.utcnow().strftime('%Y%m%d%H%M%S')}"

            # Valider la branche
            count = self.spark.sql(f"""
                SELECT COUNT(*) as cnt FROM {table_name}.branch_{branch_name}
            """).collect()[0]['cnt']

            elapsed = time.time() - start_time

            # Nettoyer
            self.spark.sql(f"ALTER TABLE {table_name} DROP BRANCH {branch_name}")

            success = elapsed <= max_rto_seconds and count > 0
            return self._record_result(
                table_name, "rollback", success,
                f"Durée: {elapsed:.2f}s, Enregistrements: {count}, Snapshot:
{target_snapshot}",
                elapsed
            )
```

```
        except Exception as e:
            elapsed = time.time() - start_time
            return self._record_result(table_name, "rollback", False, str(e), elapsed)

    def test_register_table(self, table_path: str, metadata_file: str,
                            max_rto_seconds: int) -> Dict:
        """Teste la récupération par ré-enregistrement de table."""
        start_time = time.time()

        try:
            # Enregistrer la table avec un nouveau nom temporaire
            test_table_name = f"dr_test_{datetime.utcnow().strftime('%Y%m%d_%H%M%S')}"

            self.spark.sql(f"""
                CALL iceberg.system.register_table(
                    table => 'dr_catalog.{test_table_name}',
                    metadata_file => '{metadata_file}'
                )
            """)

            # Valider que la table est accessible
            count = self.spark.sql(f"""
                SELECT COUNT(*) as cnt FROM dr_catalog.{test_table_name}
            """).collect()[0]['cnt']

            # Vérifier les métadonnées
            schema_cols = self.spark.sql(f"""
                DESCRIBE dr_catalog.{test_table_name}
            """).count()

            elapsed = time.time() - start_time

            # Nettoyer
            self.spark.sql(f"DROP TABLE dr_catalog.{test_table_name}")

            success = elapsed <= max_rto_seconds and count > 0
            return self._record_result(
                table_path, "register", success,
                f"Durée: {elapsed:.2f}s, Enregistrements: {count}, Colonnes:
{schema_cols}",
                elapsed
            )

        except Exception as e:
            elapsed = time.time() - start_time
            return self._record_result(table_path, "register", False, str(e), elapsed)

    def test_time_travel_query(self, table_name: str, hours_back: int = 48) -> Dict:
        """Teste la capacité de requête time travel."""
        start_time = time.time()

        try:
            target_time = datetime.utcnow() - timedelta(hours=hours_back)

            # Exécuter une requête time travel
            result = self.spark.sql(f"""
                SELECT COUNT(*) as cnt
                FROM {table_name}
```

```
                FOR TIMESTAMP AS OF TIMESTAMP '{target_time.isoformat()}'
            """).collect()[0]['cnt']

            elapsed = time.time() - start_time

            success = result > 0
            return self._record_result(
                table_name, "time_travel", success,
                f"Durée: {elapsed:.2f}s, Enregistrements à t-{hours_back}h: {result}",
                elapsed
            )

        except Exception as e:
            elapsed = time.time() - start_time
            return self._record_result(table_name, "time_travel", False, str(e), elapsed)

    def _record_result(self, table: str, test_type: str, success: bool,
                       message: str, duration: float) -> Dict:
        """Enregistre le résultat d'un test."""
        result = {
            'table': table,
            'test_type': test_type,
            'success': success,
            'message': message,
            'duration_seconds': round(duration, 2),
            'timestamp': datetime.utcnow().isoformat()
        }
        self.results.append(result)
        return result

    def run_full_test_suite(self, critical_tables: List[Dict]) -> Dict:
        """Exécute la suite complète de tests DR."""
        print(f"Démarrage des tests DR à {datetime.utcnow().isoformat()}")
        print(f"Tables à tester : {len(critical_tables)}")

        for table_config in critical_tables:
            table_name = table_config['name']
            max_rto = table_config.get('max_rto_seconds', 300)
            metadata_file = table_config.get('metadata_file')

            print(f"\nTest de {table_name}...")

            # Test 1: Rollback de snapshot
            self.test_snapshot_rollback(table_name, max_rto)

            # Test 2: Time travel
            self.test_time_travel_query(table_name)

            # Test 3: Register table (si metadata_file fourni)
            if metadata_file:
                self.test_register_table(table_name, metadata_file, max_rto)

        return self.generate_report()

    def generate_report(self) -> Dict:
        """Génère un rapport de test DR complet."""
        total = len(self.results)
        passed = sum(1 for r in self.results if r['success'])
        failed = total - passed
```

```
        avg_duration = sum(r['duration_seconds'] for r in self.results) / total if total >
0 else 0
        max_duration = max(r['duration_seconds'] for r in self.results) if self.results
else 0

        report = {
            'summary': {
                'generated_at': datetime.utcnow().isoformat(),
                'total_tests': total,
                'passed': passed,
                'failed': failed,
                'success_rate': f"{(passed/total)*100:.1f}%" if total > 0 else "N/A",
                'average_duration_seconds': round(avg_duration, 2),
                'max_duration_seconds': round(max_duration, 2)
            },
            'by_test_type': {},
            'failures': [r for r in self.results if not r['success']],
            'details': self.results
        }

        # Agrégation par type de test
        for test_type in ['rollback', 'register', 'time_travel']:
            type_results = [r for r in self.results if r['test_type'] == test_type]
            if type_results:
                report['by_test_type'][test_type] = {
                    'total': len(type_results),
                    'passed': sum(1 for r in type_results if r['success']),
                    'avg_duration': round(
                        sum(r['duration_seconds'] for r in type_results) /
len(type_results), 2
                    )
                }

        return report
```

> **Performance** La réplication inter-région peut avoir un impact significatif sur les performances d'écriture si elle est synchrone. Pour les tables à haut débit d'écriture, privilégier une réplication asynchrone avec un objectif RPO de 15-60 minutes, combinée à des snapshots horodatés réguliers permettant une récupération à un point précis dans le temps.

Plan de récupération après sinistre

Un plan de DR documenté et testé est essentiel. Ce plan doit inclure les éléments suivants.

Inventaire des actifs : Liste de toutes les tables Iceberg, leur classification de criticité, et les chemins des fichiers metadata.json correspondants.

Procédures de récupération : Scripts et runbooks pour chaque scénario de défaillance identifié, avec les commandes exactes à exécuter.

Rôles et responsabilités : Identification des personnes autorisées à déclencher une récupération et des équipes responsables de chaque composant.

Tests réguliers : Simulation de sinistres et exécution des procédures de récupération dans un environnement de test, au minimum trimestriellement.

```
# Runbook de récupération — Suppression accidentelle de données

## Prérequis
- Accès administrateur au cluster Spark
- Identifiants du catalogue Iceberg
- Connaissance de l'heure approximative de l'incident

## Étapes

1. **Identifier le snapshot cible**
   ```sql
   SELECT snapshot_id, committed_at, operation, summary
   FROM <table>.snapshots
   WHERE committed_at < '<heure_incident>'
   ORDER BY committed_at DESC
   LIMIT 10;
```

2. **Valider le snapshot** (vérifier que les données attendues sont présentes)

```
SELECT COUNT(*), MAX(updated_at)
FROM <table> FOR VERSION AS OF <snapshot_id>;
```

3. **Exécuter le rollback**

```
CALL iceberg.system.rollback_to_snapshot('<table>', <snapshot_id>);
```

4. **Valider la récupération**

```
SELECT COUNT(*), MAX(updated_at) FROM <table>;
```

5. **Notifier les parties prenantes**

- Informer l'équipe data engineering
- Créer un ticket d'incident
- Documenter la cause racine

Escalade

Si la récupération échoue, contacter le DBA de garde au [numéro].

```
### Réplication multi-région et haute disponibilité

Pour les organisations exigeant une haute disponibilité (HA) avec des RTO de quelques
minutes, une architecture multi-région active-passive ou active-active est nécessaire.

**Architecture active-passive** : Une région primaire traite toutes les écritures, avec
réplication asynchrone vers une région secondaire. En cas de sinistre, la région
secondaire est promue en primaire. Cette architecture offre un bon compromis entre coût et
```
```

```
résilience.

**Architecture active-active** : Les deux régions acceptent les écritures, avec résolution
des conflits. Cette architecture est complexe à implémenter avec Iceberg en raison des
garanties ACID, et nécessite généralement une couche de coordination externe.

Les catalogues modernes comme Apache Polaris et Nessie offrent des fonctionnalités de
réplication qui facilitent la synchronisation des métadonnées entre régions. Combinées à
la réplication de stockage objet, ces fonctionnalités permettent de construire des
architectures HA sophistiquées.

### Conception d'une architecture multi-région

L'architecture multi-région pour un lakehouse Iceberg requiert une planification
minutieuse des composants de stockage, de catalogue et de calcul.

**Composant stockage** : Le stockage objet constitue la fondation. AWS S3, Azure Blob
Storage et Google Cloud Storage offrent tous des options de réplication inter-région. La
configuration dépend des exigences RPO.

- **Réplication S3 Cross-Region** : Latence typique de 15 minutes pour la réplication.
Convient pour RPO > 15 minutes.
- **S3 Multi-Region Access Points** : Permet des écritures automatiquement routées vers la
région la plus proche. Convient pour les architectures actif-actif.
- **Azure Geo-Redundant Storage (GRS)** : Réplication synchrone vers une région
secondaire. RPO proche de zéro mais surcoût significatif.

```bash
# Configuration AWS CLI pour activer la réplication CRR
aws s3api put-bucket-replication \
    --bucket lakehouse-primary \
    --replication-configuration file://replication-config.json

# Vérification du statut de réplication
aws s3api get-bucket-replication-metrics \
    --bucket lakehouse-primary \
    --id ReplicationRule1
```

Composant catalogue : Le catalogue doit être répliqué pour permettre la découverte des tables dans la région DR.

- **Nessie** : Supporte la réplication via export/import des commits. La communauté travaille sur une réplication native.
- **Apache Polaris** : Offre des API de synchronisation pour les environnements multi-région.
- **Hive Metastore** : Peut être répliqué via la réplication de la base de données sous-jacente (MySQL, PostgreSQL).

```python
# Script de synchronisation Nessie vers région DR
import requests
from datetime import datetime

class NessieReplicator:
    """Réplique les commits Nessie vers une région DR."""

    def __init__(self, source_uri: str, target_uri: str):
        self.source_uri = source_uri
        self.target_uri = target_uri
```
```

```
        self.last_synced_commit = None

    def get_commits(self, branch: str = "main", since_commit: str = None) -> list:
        """Récupère les commits depuis la source."""
        url = f"{self.source_uri}/api/v2/trees/{branch}/log"
        params = {}
        if since_commit:
            params['filter'] = f"commit.hash != '{since_commit}'"

        response = requests.get(url, params=params)
        return response.json().get('logEntries', [])

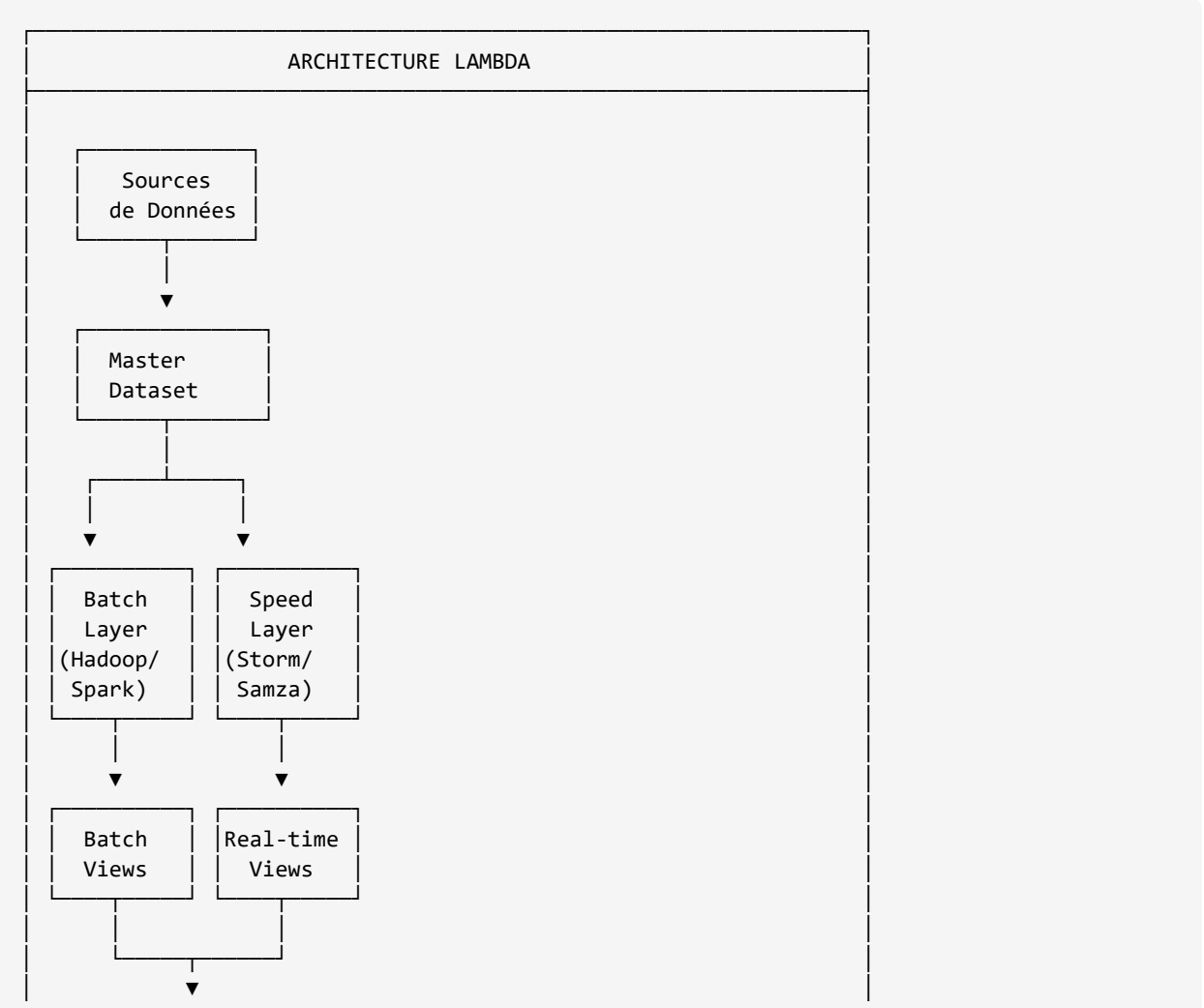
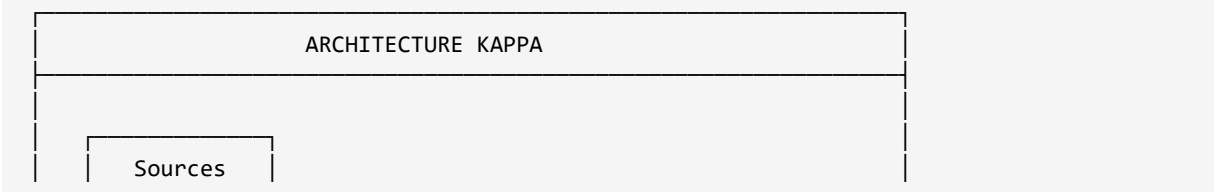
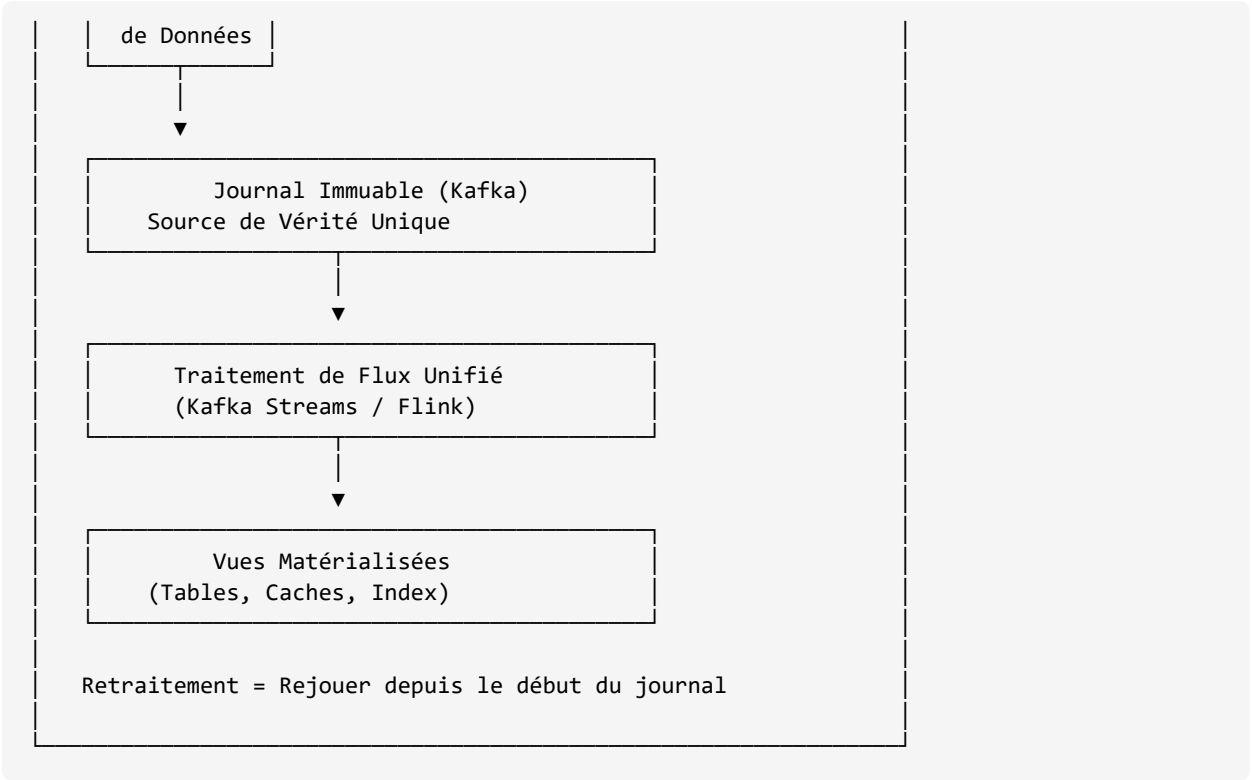
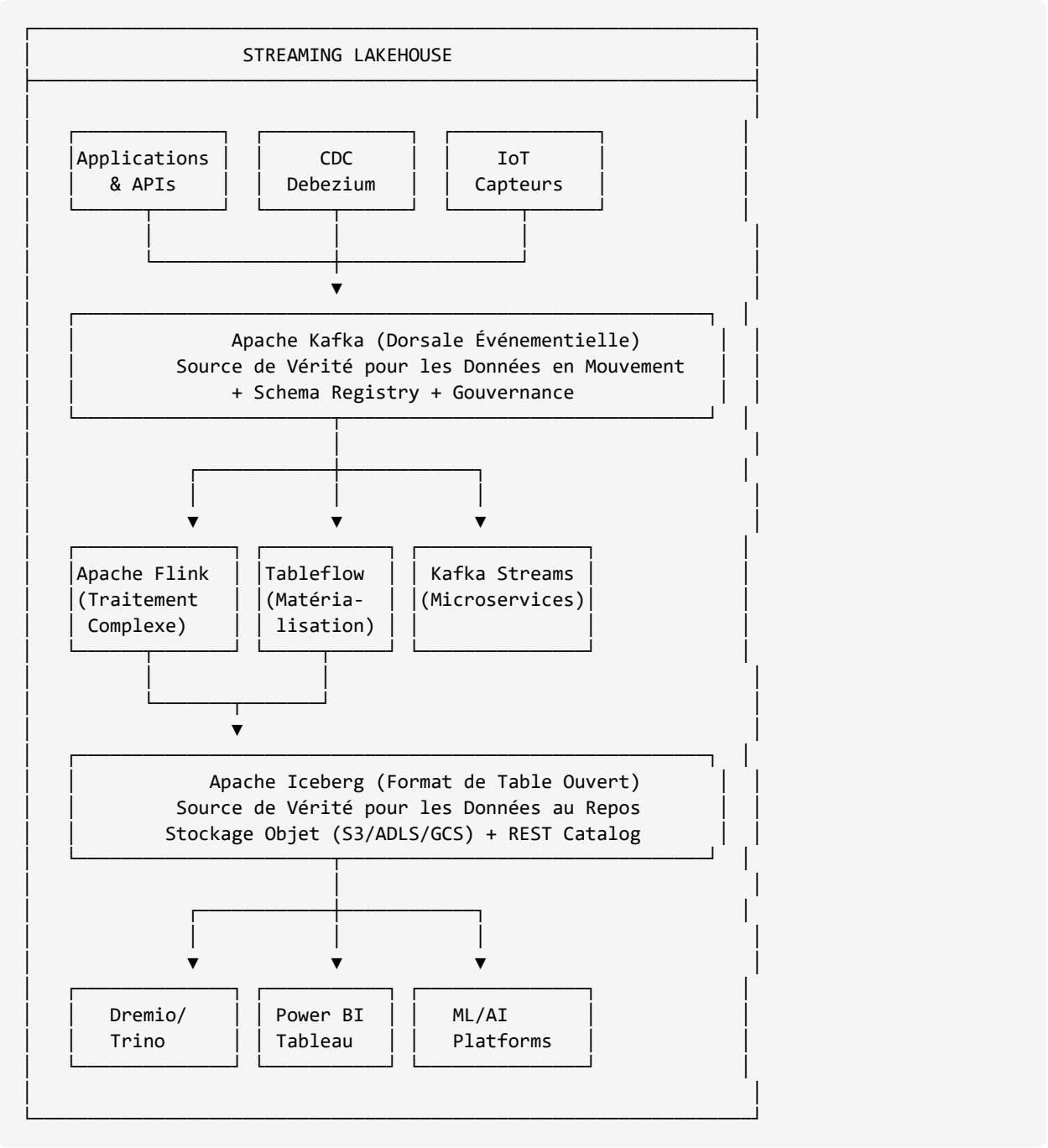
    def replicate_commits(self, commits: list):
        """Réplique les commits vers la cible."""
        for commit in reversed(commits):  # Du plus ancien au plus récent
            # Récupérer les opérations du commit
            operations = self.get_commit_operations(commit['commitMeta']['hash'])

            # Appliquer à la cible
            self.apply_operations(operations, commit['commitMeta'])

            self.last_synced_commit = commit['commitMeta']['hash']

    def get_commit_operations(self, commit_hash: str) -> list:
        """Récupère les opérations d'un commit."""
        url = f"{self.source_uri}/api/v2/trees/main/diff"
        params = {'from': f'{commit_hash}^', 'to': commit_hash}
        response = requests.get(url, params=params)
        return response.json().get('diffs', [])

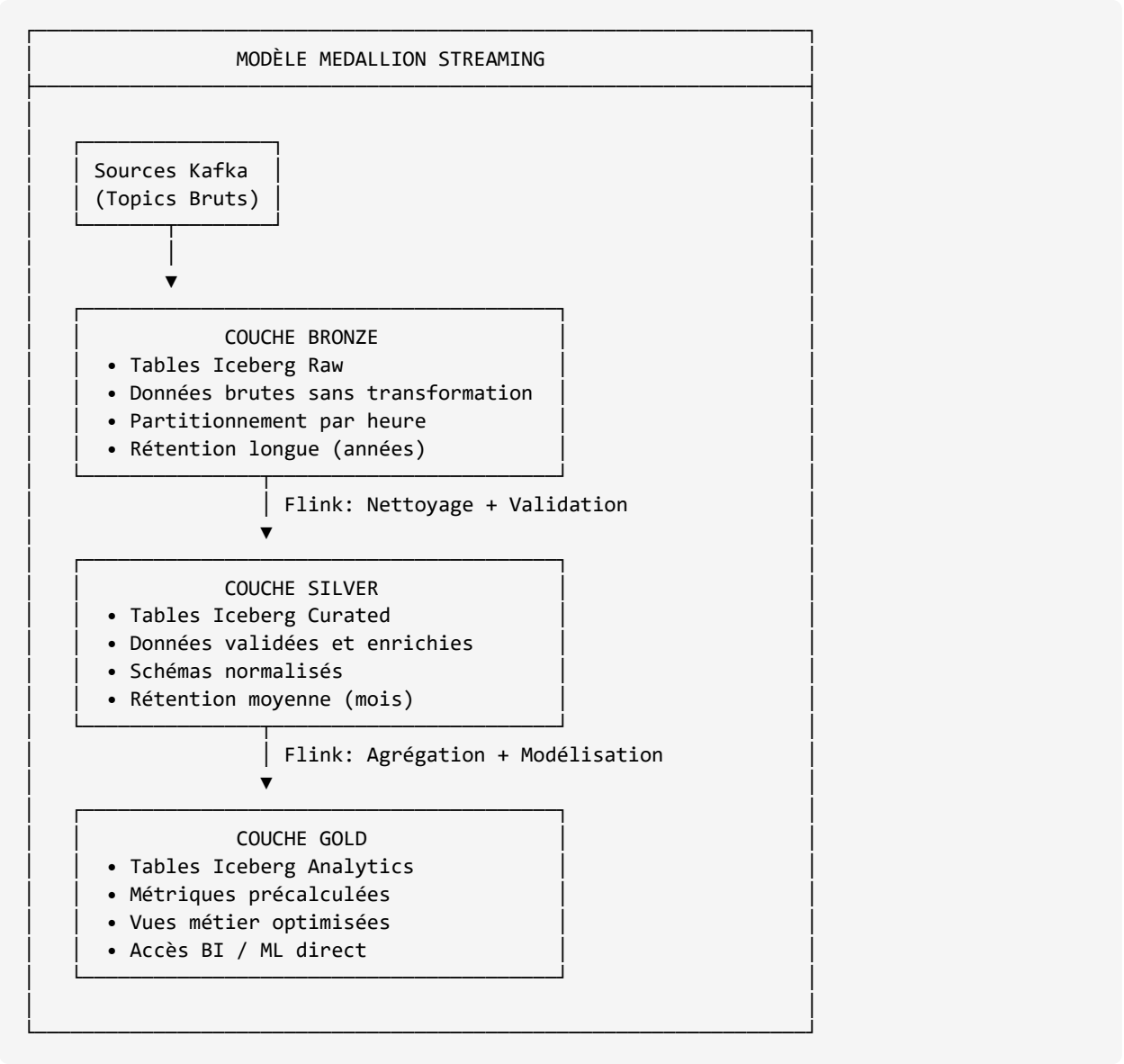
    def apply_operations(self, operations: list, commit_meta: dict):
        """Applique les opérations à la cible."""
        url = f"{self.target_uri}/api/v2/trees/main/commit"
        payload = {
            'commitMeta': {
                'message': f"[REPLICATED] {commit_meta.get('message', '')}",
                'author': commit_meta.get('author'),
                'properties': {
                    'source_commit': commit_meta.get('hash'),
                    'replicated_at': datetime.utcnow().isoformat()
                }
            },
            'operations': operations
        }
        response = requests.post(url, json=payload)
        response.raise_for_status()

    def sync(self, branch: str = "main"):
        """Exécute une synchronisation incrémentale."""
        commits = self.get_commits(branch, self.last_synced_commit)
        if commits:
            print(f"Réplication de {len(commits)} commits...")
            self.replicate_commits(commits)
            print(f"Synchronisation terminée. Dernier commit: {self.last_synced_commit}")
        else:
            print("Aucun nouveau commit à répliquer.")
```

Composant calcul : Les clusters Spark, Trino ou Dremio doivent être déployés dans chaque région avec accès au stockage et au catalogue locaux.

Considérations de coûts pour la DR

La récupération après sinistre engendre des coûts significatifs qu'il convient d'optimiser selon les exigences réelles.

Composant	Coût mensuel typique (100 TB)	Stratégie d'optimisation
Réplication stockage	500-2000 $	Répliquer seulement les tables critiques
Stockage DR	1000-3000 $	Utiliser des classes de stockage économiques (S3 IA)
Infrastructure de test DR	200-500 $	Clusters éphémères pour les tests
Bande passante inter-région	100-500 $	Compression des données répliquées

Stratégies d'optimisation des coûts :

1. **Classification des données** : Ne pas répliquer toutes les tables avec la même stratégie. Les données d'archivage peuvent tolérer un RPO plus long et utiliser une réplication asynchrone moins coûteuse.
2. **Réplication sélective** : Répliquer uniquement les métadonnées pour les tables volumineuses mais facilement reconstituables. Les fichiers de données peuvent être régénérés depuis les sources en cas de sinistre.
3. **Compression et déduplication** : Les outils de réplication peuvent compresser les données en transit, réduisant les coûts de bande passante.
4. **Classes de stockage DR** : Utiliser S3 Intelligent-Tiering ou S3 Glacier Instant Retrieval pour le stockage DR, réduisant les coûts tout en maintenant l'accessibilité.

```
# Exemple de politique de réplication différenciée par criticité
replication_policies = {
    'critical': {
        'tables': ['customer_360', 'transactions', 'orders'],
        'storage_class': 'STANDARD',
        'replication_time_minutes': 15,
        'retain_snapshots_days': 30
    },
    'important': {
        'tables': ['product_catalog', 'inventory', 'pricing'],
        'storage_class': 'STANDARD_IA',
        'replication_time_minutes': 60,
        'retain_snapshots_days': 14
    },
    'standard': {
        'tables': ['analytics_*', 'reporting_*'],
        'storage_class': 'INTELLIGENT_TIERING',
        'replication_time_minutes': 240,
        'retain_snapshots_days': 7
    },
    'archive': {
        'tables': ['historical_*', 'audit_logs_*'],
        'storage_class': 'GLACIER_IR',
        'replication_time_minutes': 1440,  # 24 heures
```

```
        'retain_snapshots_days': 90
    }
}
```

> **Étude de cas : Banque Nationale du Canada** *Secteur* : Services financiers *Défi* : Répondre aux exigences du Bureau du surintendant des institutions financières (BSIF) pour la continuité des activités, avec un RTO de 2 heures et un RPO de 15 minutes pour les données critiques. *Solution* : Architecture multi-région active-passive avec lakehouse Iceberg principal à Toronto et DR à Montréal. Réplication S3 Cross-Region Replication pour les fichiers de données. Catalogue Nessie avec réplication des commits vers la région DR. Tests de basculement trimestriels. *Résultats* : RTO effectif de 45 minutes lors du dernier test de basculement. RPO de 12 minutes garanti par la fréquence de réplication. Conformité BSIF attestée.

IV.11.4 Résumé

L'opérationnalisation d'Apache Iceberg transforme une architecture de données prometteuse en une plateforme de production fiable et résiliente. Ce chapitre a exploré les trois piliers de cette transformation : l'orchestration, l'audit et la récupération après sinistre.

Orchestration du lakehouse

L'orchestration constitue le système nerveux central du lakehouse, coordonnant les pipelines d'ingestion, de transformation et de maintenance. Les orchestrateurs modernes se divisent en deux catégories : les outils orientés tâches comme Apache Airflow, et les outils orientés actifs comme Dagster.

Pour les lakehouses Iceberg, l'orchestration orientée actifs présente des avantages significatifs en termes de lignée, d'observabilité et de gestion des partitions. Les modèles d'orchestration incluent l'approche par horaire, l'approche événementielle et l'hybride, chacun adapté à des exigences de latence et de complexité différentes.

Les opérations de maintenance Iceberg (compaction, expiration des snapshots, nettoyage des orphelins) doivent être intégrées dans les pipelines orchestrés avec des stratégies conditionnelles basées sur les métriques des tables.

Audit du lakehouse

L'audit répond aux exigences de conformité réglementaire tout en fournissant une traçabilité essentielle pour le diagnostic et l'analyse forensique. Les tables de métadonnées Iceberg offrent une base solide pour l'audit, capturant l'historique complet des modifications.

Une architecture d'audit complète comprend les journaux d'accès (générés par les moteurs de requête), les journaux de modification (via les snapshots Iceberg) et la lignée des données (maintenue par les catalogues et les orchestrateurs).

L'observabilité transforme les données d'audit en informations actionnables, avec des métriques clés à surveiller incluant la fragmentation des fichiers, la latence d'ingestion et les taux d'échec.

Récupération après sinistre

La stratégie de DR doit être alignée sur les objectifs RTO et RPO définis pour chaque classe de données. Iceberg offre des capacités natives de récupération grâce à l'immutabilité des fichiers, les snapshots comme points de récupération et les procédures intégrées (rollback, register_table, rewrite_table_path).

Les scénarios de récupération couverts incluent les erreurs humaines (rollback vers un snapshot antérieur), la corruption du catalogue (ré-enregistrement des tables) et la restauration vers un nouvel emplacement (réécriture des chemins).

Pour les exigences de haute disponibilité, les architectures multi-région active-passive ou active-active peuvent être implémentées en combinant la réplication de stockage objet et la réplication des métadonnées du catalogue.

Principes directeurs

L'opérationnalisation réussie d'un lakehouse Iceberg repose sur plusieurs principes directeurs :

1. **Automatisation systématique** : Toute opération répétitive doit être automatisée et orchestrée, éliminant les interventions manuelles sources d'erreurs.
2. **Observabilité native** : Les métriques, journaux et traces doivent être collectés et corrélés pour permettre un diagnostic rapide et une détection proactive des anomalies.
3. **Défense en profondeur** : La protection des données repose sur plusieurs couches : snapshots pour la récupération rapide, sauvegardes pour la protection contre la corruption, réplication pour la continuité régionale.
4. **Tests réguliers** : Les procédures de récupération doivent être testées régulièrement dans des conditions réalistes pour valider leur efficacité.
5. **Documentation vivante** : Les runbooks et procédures doivent être maintenus à jour et accessibles à toutes les équipes concernées.

Le chapitre suivant explorera l'évolution vers le *Streaming Lakehouse*, où l'intégration d'Apache Kafka avec Apache Iceberg permet de traiter les données en temps réel tout en maintenant les garanties de cohérence et la gouvernance du lakehouse.

Points clés à retenir

- L'orchestration orientée actifs (Dagster, Prefect) s'aligne naturellement avec l'architecture Iceberg centrée sur les tables et les snapshots
- Les tables de métadonnées Iceberg (`snapshots` , `history` , `files` , `manifests`) constituent des sources d'audit natives nécessitant un enrichissement contextuel
- Le *time travel* Iceberg permet une récupération quasi instantanée des erreurs humaines sans restauration de fichiers
- La procédure `register_table` permet de reconstruire un catalogue corrompu à partir des fichiers de métadonnées
- La réplication multi-région combine la réplication du stockage objet et la synchronisation des métadonnées du catalogue
- Les tests de récupération réguliers sont essentiels pour valider les procédures et former les équipes

Références

- Apache Iceberg Documentation (2025). *Maintenance Operations*. https://iceberg.apache.org/docs/latest/maintenance/
- Dagster (2025). *Building a Better Lakehouse: From Airflow to Dagster*. https://dagster.io/blog/building-a-better-lakehouse
- Dremio (2025). *Disaster Recovery for Apache Iceberg Tables*. https://www.dremio.com/blog/disaster-recovery-for-apache-iceberg-tables
- IOMETE (2025). *Iceberg Disaster Recovery*. https://iomete.com/resources/blog/iceberg-disaster-recovery
- PracData (2025). *State of Open Source Workflow Orchestration Systems 2025*. https://www.pracdata.io/p/state-of-workflow-orchestration-ecosystem-2025
- Atlan (2025). *Apache Iceberg Tables Governance: A Practical Guide*. https://atlan.com/know/iceberg/apache-iceberg-table-governance/

Chapitre IV.12 - L'ÉVOLUTION VERS LE STREAMING LAKEHOUSE

Introduction

L'histoire des architectures de données est jalonnée de compromis entre latence et complétude, entre fraîcheur et fiabilité. Pendant plus d'une décennie, les organisations ont jonglé avec des systèmes parallèles : d'un côté, les pipelines de traitement par lots (batch) offrant une vue complète mais décalée des données; de l'autre, les flux en temps réel capturant l'immédiateté au prix de la complexité. L'architecture Lambda, proposée par Nathan Marz en 2011, a tenté de réconcilier ces deux mondes en les faisant coexister. Mais cette cohabitation forcée a engendré une dette technique considérable : duplication de code, divergence des résultats, et coûts opérationnels exponentiels.

Le Streaming Lakehouse représente l'aboutissement d'une quête de plus de quinze ans vers l'unification des données en mouvement et au repos. En combinant Apache Kafka comme colonne vertébrale événementielle avec Apache Iceberg comme format de table ouvert, les organisations peuvent enfin construire une architecture où chaque événement capturé devient instantanément disponible pour l'analyse, l'intelligence artificielle et les applications opérationnelles, sans duplication ni transformation redondante.

Ce chapitre retrace l'évolution architecturale qui a conduit au Streaming Lakehouse moderne. Nous examinerons d'abord les limites des architectures Lambda et Kappa qui ont dominé la dernière décennie. Puis nous explorerons le rôle central de Confluent et Apache Kafka dans cette convergence, avec une attention particulière portée à Tableflow, la fonctionnalité qui matérialise les topics Kafka directement en tables Iceberg. Enfin, nous présenterons les patterns d'implémentation et les considérations pratiques pour les organisations canadiennes qui entreprennent cette transformation.

IV.12.1 De l'Architecture Lambda au Streaming Lakehouse

Les Origines : Le Défi du Big Data

Au tournant des années 2010, l'explosion des volumes de données a confronté les architectes à un dilemme fondamental. Les systèmes traditionnels de traitement par lots, hérités de l'ère MapReduce et Hadoop, excellaient dans le traitement exhaustif de grands volumes historiques. Ils offraient des garanties de complétude et de cohérence que les systèmes transactionnels ne pouvaient égaler à grande échelle. Cependant, leur latence intrinsèque, mesurée en heures voire en jours, les rendait inadaptés aux exigences croissantes de réactivité.

Parallèlement, les premiers systèmes de traitement de flux comme Storm (développé chez Twitter) et S4 (Yahoo) démontraient qu'il était possible de traiter des événements avec des latences de l'ordre de la milliseconde. Mais ces systèmes sacrifiaient souvent les garanties de cohérence et la capacité de retraitement historique qui caractérisaient le batch.

L'Architecture Lambda : Une Réconciliation Imparfaite

Face à ce dilemme, Nathan Marz a proposé en 2011 l'architecture Lambda, une approche hybride qui fait coexister deux chemins de traitement parallèles :

La couche batch (Batch Layer) traite l'intégralité des données historiques selon des cycles réguliers (typiquement journaliers ou horaires). Elle utilise des technologies comme Hadoop MapReduce ou Apache Spark pour produire des « vues batch » exhaustives et cohérentes. Cette couche constitue la source de vérité, capable de corriger toute erreur introduite par la couche temps réel.

La couche vitesse (Speed Layer) traite les données en temps réel à mesure de leur arrivée. Elle produit des « vues temps réel » qui compensent la latence de la couche batch. Ces vues sont approximatives et temporaires, destinées à être remplacées par les résultats batch une fois disponibles.

La couche de service (Serving Layer) fusionne les vues batch et temps réel pour présenter une vue unifiée aux consommateurs. Elle gère la complexité de la réconciliation entre des résultats potentiellement divergents.



Les Limites Fondamentales de Lambda

Malgré son élégance conceptuelle, l'architecture Lambda souffre de limitations profondes qui ont progressivement conduit à son abandon par les organisations les plus avancées.

La duplication du code constitue le problème le plus immédiat. La même logique métier doit être implémentée deux fois : une version batch (souvent en Scala/Spark) et une version streaming (Storm, Samza, ou autre). Ces implémentations divergent inévitablement au fil du temps, introduisant des incohérences subtiles entre les résultats batch et temps réel.

La divergence des résultats découle de cette duplication. Les vues batch et temps réel peuvent produire des résultats différents pour les mêmes données, créant des situations où les utilisateurs observent des « sauts » dans les métriques lors de la réconciliation.

La complexité opérationnelle est considérable. L'équipe doit maîtriser et opérer deux écosystèmes technologiques distincts, chacun avec ses propres modes de défaillance, patterns de déploiement et exigences de monitoring.

Les coûts d'infrastructure sont doublés. Les données sont stockées et traitées deux fois, multipliant les besoins en calcul et en stockage.

> **Migration**
> *De* : Architecture Lambda avec Hadoop + Storm
> *Vers* : Streaming Lakehouse unifié
> *Stratégie* : Migration incrémentale en commençant par les cas d'usage les plus critiques en latence. Maintenir Lambda en parallèle pendant la transition, puis décommissionner progressivement la couche batch.

L'Architecture Kappa : La Simplification par le Flux

En 2014, Jay Kreps, cofondateur d'Apache Kafka, a proposé l'architecture Kappa comme alternative radicale à Lambda. Son principe fondateur est simple : traiter toutes les données comme un flux, éliminant ainsi la distinction entre batch et temps réel.

Dans Kappa, le journal d'événements immuable (typiquement un cluster Kafka) devient la source de vérité unique. Toutes les transformations sont exprimées comme des applications de traitement de flux. Lorsqu'une correction ou une évolution de la logique est nécessaire, l'application est simplement rejouée depuis le début du journal pour reconstruire l'état correct.





L'architecture Kappa offre des avantages significatifs par rapport à Lambda. L'élimination de la duplication de code simplifie drastiquement le développement et la maintenance. La source de vérité unique garantit la cohérence des résultats. Les coûts opérationnels sont réduits par l'utilisation d'une seule pile technologique.

Cependant, Kappa présente ses propres limitations. Le retraitement depuis le début du journal peut s'avérer coûteux en temps et en ressources pour des historiques volumineux. Les résultats intermédiaires stockés dans Kafka ne sont pas directement interrogeables par SQL, limitant l'accès aux analystes et aux outils d'intelligence d'affaires. Enfin, le stockage prolongé dans Kafka peut devenir onéreux comparé aux solutions de stockage objet.

L'Émergence du Data Lakehouse

Le concept de Data Lakehouse, introduit vers 2017, a apporté une réponse partielle à ces limitations. En combinant la scalabilité économique des lacs de données avec les garanties transactionnelles des entrepôts de données, les formats de table ouverts comme Apache Iceberg, Delta Lake et Apache Hudi ont transformé les lacs de données en systèmes fiables et interrogeables.

Ces formats introduisent des capacités essentielles pour la gestion de données à l'échelle de l'entreprise. Les transactions ACID garantissent la cohérence des lectures et écritures concurrentes. L'évolution de schéma permet de modifier la structure des tables sans réécriture complète. Le partitionnement masqué (hidden partitioning) optimise les performances de requête sans exposer la complexité aux utilisateurs. Le time travel permet d'interroger les données telles qu'elles existaient à un instant donné.

Cependant, les Lakehouses traditionnels restent fondamentalement orientés batch. Leurs optimisations ciblent les requêtes analytiques sur de grands volumes statiques plutôt que l'ingestion continue à faible latence. L'intégration avec les systèmes de streaming nécessite des pipelines ETL additionnels, réintroduisant la complexité que Kappa cherchait à éliminer.

Le Streaming Lakehouse : La Convergence

Le Streaming Lakehouse représente la synthèse de ces évolutions architecturales. Il unifie les forces de Kappa (traitement unifié, source de vérité événementielle) avec celles du Lakehouse (stockage économique, interrogation SQL, gouvernance des données) en une architecture cohérente.



Les caractéristiques distinctives du Streaming Lakehouse incluent :

L'ingestion unifiée : Les données transitent par Kafka quelle que soit leur source (applications, CDC, IoT, fichiers batch). Cette normalisation simplifie la gouvernance et assure la traçabilité complète.

La matérialisation automatique : Les topics Kafka sont automatiquement matérialisés en tables Iceberg, éliminant les pipelines ETL manuels. Des solutions comme Tableflow (Confluent) ou le Flink Dynamic Iceberg Sink gèrent cette conversion de manière transparente.

Le stockage unique : Les données sont stockées une seule fois dans un format ouvert (Parquet sur stockage objet), accessible par tous les moteurs de consommation sans duplication.

L'interrogation universelle : Les mêmes tables sont accessibles via des moteurs de streaming (Flink), des moteurs SQL analytiques (Trino, Dremio), des outils d'intelligence d'affaires (Power BI, Tableau), et des plateformes ML/AI.

Comparaison des Architectures

Le tableau suivant synthétise les caractéristiques des trois architectures évoquées :

Critère	Lambda	Kappa	Streaming Lakehouse
Duplication de code	Élevée (2 implémentations)	Aucune	Aucune
Latence	Mixte (temps réel + batch)	Temps réel	Temps réel + batch unifié
Interrogation SQL	Via couche de service	Limitée	Native sur Iceberg
Coût de stockage	Élevé (duplication)	Moyen (Kafka long terme)	Optimal (stockage objet)
Retraitement	Via couche batch	Rejeu du journal	Rejeu + time travel
Gouvernance	Fragmentée	Sur Kafka	Unifiée (Kafka + Iceberg)
Complexité opérationnelle	Élevée	Moyenne	Moyenne
Maturité	Mature (2011)	Mature (2014)	Émergente (2023+)

> **Performance**
> Les organisations qui ont migré de Lambda vers un Streaming Lakehouse rapportent typiquement une réduction de 40 à 60 % des coûts d'infrastructure, principalement grâce à l'élimination de la duplication des données et du traitement. La latence de bout en bout pour les cas d'usage analytiques passe de T+1 jour à quelques minutes.

Le Pattern « Shift Left »

L'architecture Shift Left, décrite par Kai Waehner de Confluent, représente une évolution philosophique importante. Plutôt que de considérer la gouvernance et la qualité des données comme des préoccupations en aval (dans l'entrepôt ou le lakehouse), cette approche les déplace « vers la gauche », c'est-à-dire plus près de la source.

Dans un Streaming Lakehouse adoptant Shift Left, les transformations de données, les validations de qualité et l'enrichissement sémantique s'effectuent dans la couche de streaming (Kafka + Flink) plutôt que dans des processus ETL tardifs. Les données arrivent dans le Lakehouse déjà structurées, validées et conformes aux contrats de schéma définis dans le Schema Registry.

Ce pattern offre plusieurs avantages pour les organisations canadiennes soumises à des réglementations strictes (Loi 25, LPRPDE) :

- La traçabilité de chaque transformation est garantie par le journal Kafka
- Les règles de qualité sont appliquées en temps réel, détectant les anomalies immédiatement
- La gouvernance des schémas via le Schema Registry prévient les évolutions incompatibles
- Les métadonnées de lignage sont capturées automatiquement pour l'audit

L'implémentation du Shift Left avec Apache Flink illustre cette approche :

```
-- Flink SQL : Pipeline Shift Left avec validation et enrichissement
-- Les données sont nettoyées et enrichies AVANT d'atteindre le Lakehouse

-- Table source brute depuis Kafka
CREATE TABLE raw_transactions (
    transaction_id STRING,
    account_id STRING,
    amount DECIMAL(18, 2),
    currency STRING,
    merchant_name STRING,
    merchant_category STRING,
    transaction_time TIMESTAMP(3),
    WATERMARK FOR transaction_time AS transaction_time - INTERVAL '30' SECOND
) WITH (
    'connector' = 'kafka',
    'topic' = 'raw.transactions',
    'properties.bootstrap.servers' = 'kafka:9092',
    'format' = 'avro-confluent',
    'avro-confluent.url' = 'http://schema-registry:8081'
);

-- Table de référence pour enrichissement (lookup join)
CREATE TABLE merchant_categories (
    merchant_category STRING,
    category_group STRING,
    risk_level STRING,
    PRIMARY KEY (merchant_category) NOT ENFORCED
) WITH (
    'connector' = 'jdbc',
    'url' = 'jdbc:postgresql://postgres:5432/reference',
    'table-name' = 'merchant_categories',
    'lookup.cache.max-rows' = '10000',
    'lookup.cache.ttl' = '1h'
);

-- Table Iceberg cible avec données enrichies et validées
CREATE TABLE lakehouse_transactions (
    transaction_id STRING,
    account_id STRING,
    amount_cad DECIMAL(18, 2),
    original_amount DECIMAL(18, 2),
    original_currency STRING,
    merchant_name STRING,
    category_group STRING,
    risk_level STRING,
    transaction_time TIMESTAMP(3),
    processing_time TIMESTAMP(3),
    data_quality_score INT
```

```
) WITH (
    'connector' = 'iceberg',
    'catalog-name' = 'lakehouse',
    'database-name' = 'financial',
    'table-name' = 'transactions'
);

-- Pipeline de traitement avec validation et enrichissement
INSERT INTO lakehouse_transactions
SELECT
    t.transaction_id,
    t.account_id,
    CASE t.currency
        WHEN 'CAD' THEN t.amount
        WHEN 'USD' THEN t.amount * 1.36
        WHEN 'EUR' THEN t.amount * 1.47
        ELSE t.amount
    END AS amount_cad,
    t.amount AS original_amount,
    t.currency AS original_currency,
    COALESCE(TRIM(t.merchant_name), 'UNKNOWN') AS merchant_name,
    COALESCE(mc.category_group, 'OTHER') AS category_group,
    COALESCE(mc.risk_level, 'MEDIUM') AS risk_level,
    t.transaction_time,
    CURRENT_TIMESTAMP AS processing_time,
    CASE
        WHEN t.amount > 0 AND t.merchant_name IS NOT NULL THEN 100
        WHEN t.amount > 0 THEN 75
        ELSE 50
    END AS data_quality_score
FROM raw_transactions t
LEFT JOIN merchant_categories FOR SYSTEM_TIME AS OF t.transaction_time AS mc
    ON t.merchant_category = mc.merchant_category
WHERE t.amount IS NOT NULL
  AND t.transaction_id IS NOT NULL
  AND t.amount BETWEEN -1000000 AND 1000000;
```

Ce pipeline illustre les principes Shift Left :

1. **Validation à la source** : Les transactions invalides sont filtrées avant d'atteindre le Lakehouse
2. **Enrichissement en vol** : Les données de référence sont jointes en temps réel via lookup join
3. **Normalisation** : Les montants sont convertis en dollars canadiens pour uniformité
4. **Score de qualité** : Chaque enregistrement reçoit un indicateur exploitable en aval
5. **Traçabilité** : Le timestamp de traitement permet de suivre la latence du pipeline

Couches Bronze, Silver, Gold dans le Streaming Lakehouse

Le modèle Medallion (Bronze-Silver-Gold), popularisé par Databricks, s'adapte naturellement au Streaming Lakehouse. Chaque couche représente un niveau de raffinement des données :

Couche Bronze : Données brutes, telles qu'ingérées depuis les sources. Stockées avec un minimum de transformation pour préserver la fidélité. Utiles pour le retraitement et l'audit.

Couche Silver : Données nettoyées, validées et enrichies. Les duplications sont éliminées, les schémas sont normalisés, les références sont résolues.

Couche Gold : Données agrégées et modélisées pour la consommation métier. Métriques précalculées, dimensions denormalisées, vues optimisées pour les cas d'usage analytiques.



L'avantage du streaming dans ce modèle est la mise à jour continue de chaque couche. Contrairement à l'approche batch où les couches Silver et Gold sont recalculées périodiquement, le streaming permet une propagation quasi instantanée des changements à travers toutes les couches.

Gestion du Change Data Capture (CDC)

Le Change Data Capture représente un pattern fondamental pour alimenter le Streaming Lakehouse depuis les bases de données transactionnelles. Debezium, le standard open source pour le CDC, capture les modifications (INSERT, UPDATE, DELETE) depuis les journaux de transaction des bases de données et les publie comme événements Kafka.

L'intégration CDC-Iceberg requiert une attention particulière au format des opérations. Les événements Debezium contiennent non seulement la nouvelle valeur mais aussi l'ancienne valeur et le type d'opération. Apache Iceberg supporte nativement les opérations d'upsert via le mode Merge-on-Read, permettant de matérialiser fidèlement l'état des tables source.

```
# Configuration Debezium pour PostgreSQL vers Kafka
# debezium-postgres-config.yaml
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: postgres-cdc-connector
spec:
  class: io.debezium.connector.postgresql.PostgresConnector
  tasksMax: 1
  config:
    database.hostname: postgres-primary.internal
    database.port: 5432
    database.user: debezium
    database.password: ${secrets:cdc-password}
    database.dbname: production
    database.server.name: prod-db

    # Tables à capturer
    table.include.list: public.customers,public.orders,public.products

    # Configuration du format Avro
    key.converter: io.confluent.connect.avro.AvroConverter
    key.converter.schema.registry.url: http://schema-registry:8081
    value.converter: io.confluent.connect.avro.AvroConverter
    value.converter.schema.registry.url: http://schema-registry:8081

    # Options CDC
    publication.autocreate.mode: filtered
    slot.name: debezium_prod
    plugin.name: pgoutput

    # Transformations pour extraction des valeurs
    transforms: unwrap
    transforms.unwrap.type: io.debezium.transforms.ExtractNewRecordState
    transforms.unwrap.drop.tombstones: false
    transforms.unwrap.delete.handling.mode: rewrite
```

La matérialisation des flux CDC en tables Iceberg avec support des opérations DELETE utilise le mode upsert de Flink :

```
-- Table Iceberg configurée pour CDC avec upsert
CREATE TABLE iceberg_customers (
    customer_id STRING,
    email STRING,
    full_name STRING,
    created_at TIMESTAMP(3),
    updated_at TIMESTAMP(3),
    is_deleted BOOLEAN,
    PRIMARY KEY (customer_id) NOT ENFORCED
) WITH (
    'connector' = 'iceberg',
    'catalog-name' = 'lakehouse',
    'database-name' = 'crm',
    'table-name' = 'customers',
    'write.upsert.enabled' = 'true',
    'write.format.default' = 'parquet',
    'write.merge-mode' = 'merge-on-read'
```

```
);

-- Ingestion CDC avec gestion des suppressions logiques
INSERT INTO iceberg_customers
SELECT
    customer_id,
    email,
    full_name,
    created_at,
    CURRENT_TIMESTAMP AS updated_at,
    __deleted AS is_deleted
FROM cdc_customers_kafka
WHERE customer_id IS NOT NULL;
```

Migration
De : Réplication batch nocturne depuis PostgreSQL
Vers : CDC temps réel avec Debezium vers Streaming Lakehouse
Stratégie : 1. Déployer Debezium en mode snapshot pour la capture initiale 2. Basculer en mode streaming après synchronisation complète 3. Maintenir la réplication batch en parallèle pendant validation (2-4 semaines) 4. Décommissionner le batch après période de stabilisation

Architecture Delta et Streamhouse

Au-delà du Streaming Lakehouse, d'autres évolutions architecturales méritent attention. L'architecture Delta, portée par Databricks, utilise des techniques de micro-batching pour unifier batch et streaming dans un modèle de flux continu. Elle organise les données en couches Bronze (données brutes), Silver (données nettoyées) et Gold (données agrégées), chacune représentée par des tables Delta Lake.

Plus récemment, le concept de Streamhouse émerge comme une évolution du Lakehouse vers le streaming natif. Porté notamment par Ververica avec Apache Paimon, ce modèle place le traitement de flux au cœur de l'architecture plutôt qu'en périphérie. Paimon, développé par Alibaba et incubé chez Apache, offre un format de table optimisé pour l'ingestion streaming à haute fréquence avec support natif du CDC.

Ces évolutions confirment la tendance vers l'unification streaming-lakehouse, chaque approche offrant des compromis différents entre latence, coût et complexité.

IV.12.2 Le Rôle de Confluent et Kafka

Apache Kafka comme Système Nerveux Central

Apache Kafka s'est imposé comme la plateforme de streaming de facto pour l'entreprise moderne. Son modèle de journal distribué immuable offre des caractéristiques uniques qui en font le fondement idéal d'une architecture de Streaming Lakehouse.

Le découplage producteurs-consommateurs permet à chaque système de fonctionner à son propre rythme. Les producteurs publient des événements sans se soucier de la disponibilité ou de la capacité des consommateurs. Les consommateurs lisent à leur vitesse, reprenant là où ils s'étaient arrêtés en cas d'interruption.

La persistance durable garantit que chaque événement est stocké de manière fiable, avec une rétention configurable pouvant s'étendre sur des mois ou des années. Cette persistance permet le rejeu historique pour le retraitement ou la reconstruction d'état.

L'ordonnancement garanti au niveau de la partition assure que les événements sont traités dans l'ordre où ils ont été produits, propriété essentielle pour maintenir la cohérence des états dérivés.

La scalabilité horizontale permet d'absorber des volumes croissants en ajoutant simplement des brokers et des partitions, sans interruption de service.

L'Écosystème Confluent

Confluent, fondée par les créateurs de Kafka, a construit un écosystème complet autour du noyau open source. Pour le Streaming Lakehouse, plusieurs composants sont particulièrement pertinents.

Confluent Cloud offre Kafka en tant que service géré avec une séparation calcul-stockage, éliminant la complexité opérationnelle des clusters auto-gérés. Son architecture serverless permet une scalabilité élastique et un modèle de coût basé sur l'utilisation réelle.

Schema Registry gère les contrats de données entre producteurs et consommateurs. Il supporte les formats Avro, Protobuf et JSON Schema avec des règles de compatibilité configurables (backward, forward, full). Cette gouvernance des schémas est essentielle pour l'évolution contrôlée des structures de données.

Apache Flink (intégré) fournit un moteur de traitement de flux de niveau entreprise directement dans Confluent Cloud. Flink excelle dans les transformations complexes avec état, les jointures de flux, et les fenêtres temporelles.

Tableflow représente l'innovation la plus significative pour le Streaming Lakehouse. Cette fonctionnalité, disponible en disponibilité générale depuis mars 2025, permet de matérialiser automatiquement les topics Kafka en tables Apache Iceberg ou Delta Lake.

Tableflow : La Passerelle vers le Lakehouse

Tableflow élimine la complexité traditionnellement associée à l'alimentation d'un Lakehouse depuis Kafka. Avant Tableflow, les organisations devaient déployer et maintenir des pipelines Kafka Connect, des jobs Spark Streaming, ou des applications Flink dédiées pour convertir les événements Kafka en fichiers Parquet compatibles Iceberg.

L'architecture de Tableflow exploite des innovations dans la couche de stockage Kora de Confluent Cloud. Les segments Kafka sont directement convertis en fichiers Parquet, sans copie intermédiaire ni duplication des données. Un service de matérialisation de métadonnées génère automatiquement les métadonnées Iceberg et les journaux de transactions Delta Lake en s'appuyant sur le Schema Registry.

```
# Configuration Tableflow via API Confluent Cloud
# Exemple d'activation sur un topic existant

"""
Étape 1: Définir la configuration Tableflow
"""
tableflow_config = {
    "topic_name": "orders.events",
    "table_format": "ICEBERG",
    "catalog_type": "REST",
    "storage": {
        "type": "S3",
```

```
        "bucket": "s3://lakehouse-streaming/iceberg/",
        "region": "ca-central-1"
    },
    "schema_evolution": True,
    "compaction": {
        "enabled": True,
        "interval_minutes": 60,
        "target_file_size_mb": 256
    }
}

"""
Étape 2: Activer Tableflow via CLI Confluent
"""
# confluent tableflow enable orders.events \
#   --format iceberg \
#   --catalog-type rest \
#   --storage s3://lakehouse-streaming/iceberg/ \
#   --region ca-central-1

"""
Étape 3: Configuration Trino pour interroger les tables Iceberg
"""
# catalog.properties (trino/etc/catalog/iceberg_confluent.properties)
# connector.name=iceberg
# iceberg.catalog.type=rest
# iceberg.rest-catalog.uri=https://tableflow.confluent.cloud/v1
# iceberg.rest-catalog.security=OAUTH2
# iceberg.rest-catalog.oauth2.credential=<api-key>:<api-secret>
```

Les capacités clés de Tableflow incluent :

La conversion automatique des schémas : Les schémas Avro, Protobuf ou JSON enregistrés dans le Schema Registry sont automatiquement convertis en schémas Iceberg compatibles, avec gestion des types et des conversions nécessaires.

L'évolution de schéma transparente : Lorsqu'un producteur émet un événement avec un nouveau champ, Tableflow détecte le changement et fait évoluer le schéma de la table Iceberg de manière compatible.

La compaction automatique : Les petits fichiers générés par le streaming continu sont régulièrement compactés en fichiers plus grands, optimisés pour les requêtes analytiques.

L'intégration avec les catalogues : Tableflow supporte son propre REST Catalog ainsi que l'intégration avec AWS Glue, Snowflake Open Catalog, et Apache Polaris.

Patterns d'Intégration Kafka-Iceberg

Au-delà de Tableflow, plusieurs patterns permettent d'alimenter un Lakehouse Iceberg depuis Kafka, chacun avec ses compromis.

Pattern 1 : Kafka Connect avec Iceberg Sink

Kafka Connect offre une approche mature et flexible pour l'ingestion vers Iceberg. Le connecteur Iceberg Sink, disponible en open source, consomme les messages Kafka et les écrit en tables Iceberg.

```
# Configuration Kafka Connect Iceberg Sink
name=iceberg-sink-orders
connector.class=org.apache.iceberg.connect.IcebergSinkConnector
tasks.max=4

# Configuration Kafka
topics=orders.events
key.converter=io.confluent.connect.avro.AvroConverter
value.converter=io.confluent.connect.avro.AvroConverter
key.converter.schema.registry.url=http://schema-registry:8081
value.converter.schema.registry.url=http://schema-registry:8081

# Configuration Iceberg
iceberg.catalog.type=rest
iceberg.catalog.uri=http://nessie:19120/api/v1
iceberg.catalog.warehouse=s3://lakehouse/warehouse
iceberg.tables=default.orders
iceberg.tables.auto-create-enabled=true

# Configuration de commit
iceberg.control.commit.interval-ms=60000
iceberg.control.commit.records=10000
```

Ce pattern convient aux organisations qui disposent déjà d'une expertise Kafka Connect et souhaitent un contrôle fin sur la configuration. Il nécessite cependant la gestion d'un cluster Connect dédié.

Pattern 2 : Apache Flink avec Iceberg Sink

Apache Flink offre le plus haut niveau de flexibilité pour les transformations complexes avant l'écriture vers Iceberg. Le Flink Dynamic Iceberg Sink, introduit fin 2025, permet même de router dynamiquement vers plusieurs tables et de gérer l'évolution de schéma sans redémarrage.

```
-- Flink SQL : Création d'une table source Kafka
CREATE TABLE kafka_orders (
    order_id STRING,
    customer_id STRING,
    amount DECIMAL(10, 2),
    currency STRING,
    order_time TIMESTAMP(3),
    WATERMARK FOR order_time AS order_time - INTERVAL '5' SECOND
) WITH (
    'connector' = 'kafka',
    'topic' = 'orders.events',
    'properties.bootstrap.servers' = 'kafka:9092',
    'properties.group.id' = 'flink-iceberg-writer',
    'scan.startup.mode' = 'earliest-offset',
    'format' = 'avro-confluent',
    'avro-confluent.url' = 'http://schema-registry:8081'
);

-- Création de la table Iceberg cible
CREATE TABLE iceberg_orders (
    order_id STRING,
    customer_id STRING,
    amount DECIMAL(10, 2),
    currency STRING,
    order_time TIMESTAMP(3),
```

```
    processing_time TIMESTAMP(3)
) WITH (
    'connector' = 'iceberg',
    'catalog-name' = 'lakehouse',
    'catalog-type' = 'rest',
    'uri' = 'http://nessie:19120/api/v1',
    'warehouse' = 's3://lakehouse/warehouse',
    'database-name' = 'sales',
    'table-name' = 'orders'
);

-- Pipeline d'ingestion avec enrichissement
INSERT INTO iceberg_orders
SELECT
    order_id,
    customer_id,
    amount,
    currency,
    order_time,
    CURRENT_TIMESTAMP AS processing_time
FROM kafka_orders
WHERE amount > 0;
```

Pattern 3 : Tableflow (Zero-ETL)

Tableflow représente l'approche la plus simple, éliminant tout code ou configuration de pipeline. Les données Kafka sont automatiquement disponibles en format Iceberg sans intervention.

Pattern	Complexité	Flexibilité	Latence	Cas d'usage
Tableflow	Minimale	Moyenne	~Minutes	Matérialisation simple
Kafka Connect	Moyenne	Élevée	~Minutes	Intégration existante
Flink	Élevée	Maximale	Configurable	Transformations complexes

Le Flink Dynamic Iceberg Sink

Une avancée significative de 2025 est le Flink Dynamic Iceberg Sink, décrit en détail dans la documentation Apache Flink. Ce pattern résout un problème fondamental des pipelines d'ingestion traditionnels : leur incapacité à s'adapter aux changements sans redémarrage.

Dans une architecture traditionnelle, chaque topic Kafka nécessite un graphe de traitement (DAG) dédié avec un schéma figé. L'ajout d'un nouveau topic ou la modification d'un schéma requiert la modification du code et le redémarrage du job.

Le Dynamic Iceberg Sink permet au contraire un pipeline unique capable de :

- Écrire vers plusieurs tables Iceberg dynamiquement
- Créer automatiquement de nouvelles tables selon les instructions contenues dans les messages
- Évoluer le schéma des tables existantes en temps réel
- S'adapter aux changements de partitionnement sans interruption

```
// Configuration du Dynamic Iceberg Sink (exemple simplifié)
DataStream<GenericRecord> kafkaStream = env
```

```
    .fromSource(kafkaSource, WatermarkStrategy.noWatermarks(), "Kafka Source");

// Le DynamicIcebergSink route vers les tables selon les métadonnées du message
DynamicIcebergSink<GenericRecord> sink = DynamicIcebergSink.<GenericRecord>builder()
    .catalog(catalogName)
    .tableNameExtractor(record -> extractTableName(record))
    .schemaProvider(record -> extractSchema(record))
    .tableCreationBehavior(TableCreationBehavior.CREATE_IF_NOT_EXISTS)
    .schemaEvolutionBehavior(SchemaEvolutionBehavior.AUTO_EVOLVE)
    .build();

kafkaStream.sinkTo(sink);
```

Intégration avec le Schema Registry

Le Schema Registry de Confluent joue un rôle central dans le Streaming Lakehouse en assurant la cohérence des contrats de données entre les producteurs Kafka et les tables Iceberg.

Lorsqu'un producteur enregistre un schéma Avro dans le Registry, celui-ci devient la définition canonique de la structure des données. Tableflow et les connecteurs Iceberg utilisent ce schéma pour :

1. **Créer automatiquement** les tables Iceberg avec la structure correspondante
2. **Valider** que les nouveaux messages respectent le contrat
3. **Évoluer** le schéma Iceberg de manière compatible lors des changements
4. **Convertir** les types Avro vers les types Iceberg équivalents

La correspondance des types entre Avro et Iceberg est gérée automatiquement :

Type Avro	Type Iceberg
string	STRING
int	INTEGER
long	LONG
float	FLOAT
double	DOUBLE
boolean	BOOLEAN
bytes (decimal)	DECIMAL(p, s)
long (timestamp-millis)	TIMESTAMP
array	LIST
map	MAP
record	STRUCT

Gestion de l'Évolution de Schéma

L'évolution de schéma représente l'un des défis les plus complexes dans un Streaming Lakehouse. Les producteurs évoluent indépendamment des consommateurs, et les tables Iceberg doivent absorber ces changements sans interruption de service.

Le Schema Registry de Confluent offre plusieurs modes de compatibilité qui régissent les évolutions autorisées :

BACKWARD (par défaut) : Les nouveaux schémas peuvent lire les données écrites avec les anciens schémas. Autorise l'ajout de champs optionnels et la suppression de champs.

FORWARD : Les anciens schémas peuvent lire les données écrites avec les nouveaux schémas. Autorise l'ajout de champs et la suppression de champs optionnels.

FULL : Combine BACKWARD et FORWARD. Autorise uniquement l'ajout ou la suppression de champs optionnels.

NONE : Aucune vérification de compatibilité. Déconseillé en production.

```
# Exemple de configuration Schema Registry pour évolution contrôlée
from confluent_kafka.schema_registry import SchemaRegistryClient
from confluent_kafka.schema_registry.avro import AvroSerializer

# Configuration du client Schema Registry
schema_registry_config = {
    'url': 'http://schema-registry:8081',
    'basic.auth.user.info': '<api-key>:<api-secret>'
}

sr_client = SchemaRegistryClient(schema_registry_config)

# Définir la règle de compatibilité pour un sujet
sr_client.set_compatibility(
    subject_name='orders.events-value',
    level='FULL_TRANSITIVE'  # Compatibilité stricte dans les deux sens
)

# Schéma Avro v1
schema_v1 = """
{
  "type": "record",
  "name": "Order",
  "namespace": "com.example.orders",
  "fields": [
    {"name": "order_id", "type": "string"},
    {"name": "customer_id", "type": "string"},
    {"name": "amount", "type": {"type": "bytes", "logicalType": "decimal", "precision":
10, "scale": 2}},
    {"name": "order_time", "type": {"type": "long", "logicalType": "timestamp-millis"}}
  ]
}
"""

# Schéma Avro v2 : Ajout d'un champ optionnel (compatible FULL)
schema_v2 = """
{
  "type": "record",
  "name": "Order",
```

```
  "namespace": "com.example.orders",
  "fields": [
    {"name": "order_id", "type": "string"},
    {"name": "customer_id", "type": "string"},
    {"name": "amount", "type": {"type": "bytes", "logicalType": "decimal", "precision":
10, "scale": 2}},
    {"name": "order_time", "type": {"type": "long", "logicalType": "timestamp-millis"}},
    {"name": "currency", "type": ["null", "string"], "default": null}
  ]
}
"""

# Tester la compatibilité avant enregistrement
is_compatible = sr_client.test_compatibility(
    subject_name='orders.events-value',
    schema=schema_v2
)

if is_compatible:
    # Enregistrer le nouveau schéma
    schema_id = sr_client.register_schema(
        subject_name='orders.events-value',
        schema=schema_v2
    )
    print(f"Schéma v2 enregistré avec ID: {schema_id}")
else:
    print("Erreur: Schéma incompatible, évolution rejetée")
```

Lorsqu'un nouveau schéma est enregistré, Tableflow détecte automatiquement le changement et fait évoluer la table Iceberg correspondante. L'opération `ALTER TABLE ... ADD COLUMN` est exécutée de manière atomique, et les nouvelles données sont écrites avec le schéma élargi tandis que les anciennes données restent lisibles avec des valeurs NULL pour les nouveaux champs.

Intégration avec les Catalogues Externes

Tableflow supporte l'intégration avec plusieurs catalogues pour exposer les tables Iceberg aux moteurs de consommation. Chaque option présente des compromis différents :

REST Catalog Confluent (intégré) : Catalogue natif de Tableflow, accessible via l'API REST standard Iceberg. Idéal pour les déploiements centrés sur Confluent Cloud.

AWS Glue Data Catalog : Intégration native avec l'écosystème AWS (Athena, EMR, Redshift Spectrum). Recommandé pour les organisations investies dans AWS.

Snowflake Open Catalog : Permet l'accès direct depuis Snowflake sans duplication. Idéal pour les utilisateurs Snowflake souhaitant interroger des données streaming.

Apache Polaris (incubating) : Catalogue open source avec gouvernance avancée, en cours d'incubation à Apache. Option de neutralité vendeur.

```
# Configuration multi-catalogue pour Tableflow
# Exemple : Synchronisation vers AWS Glue et Snowflake Open Catalog

tableflow_multi_catalog_config = {
    "topic_name": "orders.events",
    "table_format": "ICEBERG",
```

```
    # Stockage primaire
    "storage": {
        "type": "S3",
        "bucket": "s3://streaming-lakehouse-ca/iceberg/",
        "region": "ca-central-1"
    },

    # Catalogues de synchronisation
    "catalogs": [
        {
            "type": "AWS_GLUE",
            "database": "streaming_lakehouse",
            "table_prefix": "kafka_",
            "aws_region": "ca-central-1",
            "iam_role": "arn:aws:iam::123456789:role/TableflowGlueRole"
        },
        {
            "type": "SNOWFLAKE_OPEN_CATALOG",
            "catalog_name": "streaming_data",
            "schema": "kafka_tables",
            "connection": {
                "account": "org-account.canada-central.azure",
                "warehouse": "STREAMING_WH"
            }
        }
    ],

    # Options de synchronisation
    "sync_options": {
        "sync_interval_seconds": 60,
        "sync_on_commit": True,
        "propagate_schema_changes": True
    }
}
```

Traitement Pré-Lakehouse avec Flink

Avant que les données n'atteignent le Lakehouse via Tableflow ou d'autres mécanismes, Apache Flink permet d'effectuer des transformations sophistiquées. Ce traitement « pré-lakehouse » est particulièrement utile pour :

L'agrégation en temps réel : Calculer des métriques sur des fenêtres glissantes avant persistance.

Le filtrage et routage : Diriger différents types d'événements vers différentes tables.

L'enrichissement contextuel : Joindre avec des données de référence en temps réel.

La détection d'anomalies : Identifier et marquer les données suspectes.

```
-- Flink SQL : Agrégation temps réel avant persistance Iceberg
-- Calcul de métriques par fenêtre de 5 minutes

-- Table source streaming
CREATE TABLE raw_clickstream (
    user_id STRING,
    session_id STRING,
```

```
    page_url STRING,
    action_type STRING,
    device_type STRING,
    event_time TIMESTAMP(3),
    WATERMARK FOR event_time AS event_time - INTERVAL '10' SECOND
) WITH (
    'connector' = 'kafka',
    'topic' = 'clickstream.events',
    'properties.bootstrap.servers' = 'kafka:9092',
    'format' = 'avro-confluent',
    'avro-confluent.url' = 'http://schema-registry:8081',
    'scan.startup.mode' = 'latest-offset'
);

-- Table Iceberg pour les métriques agrégées
CREATE TABLE clickstream_metrics_5min (
    window_start TIMESTAMP(3),
    window_end TIMESTAMP(3),
    device_type STRING,
    page_views BIGINT,
    unique_users BIGINT,
    unique_sessions BIGINT,
    clicks BIGINT,
    avg_session_depth DOUBLE,
    bounce_rate DOUBLE
) WITH (
    'connector' = 'iceberg',
    'catalog-name' = 'lakehouse',
    'database-name' = 'analytics',
    'table-name' = 'clickstream_metrics_5min'
);

-- Pipeline d'agrégation avec fenêtre tumbling
INSERT INTO clickstream_metrics_5min
SELECT
    TUMBLE_START(event_time, INTERVAL '5' MINUTE) AS window_start,
    TUMBLE_END(event_time, INTERVAL '5' MINUTE) AS window_end,
    device_type,
    COUNT(*) AS page_views,
    COUNT(DISTINCT user_id) AS unique_users,
    COUNT(DISTINCT session_id) AS unique_sessions,
    COUNT(CASE WHEN action_type = 'click' THEN 1 END) AS clicks,
    AVG(CAST(session_depth AS DOUBLE)) AS avg_session_depth,
    CAST(
        COUNT(CASE WHEN session_depth = 1 THEN 1 END) AS DOUBLE
    ) / NULLIF(COUNT(DISTINCT session_id), 0) AS bounce_rate
FROM (
    SELECT
        *,
        COUNT(*) OVER (
            PARTITION BY session_id
            ORDER BY event_time
            ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
        ) AS session_depth
    FROM raw_clickstream
)
GROUP BY
    TUMBLE(event_time, INTERVAL '5' MINUTE),
    device_type;
```

```
-- Table Iceberg pour les sessions individuelles (grain fin)
CREATE TABLE user_sessions (
    session_id STRING,
    user_id STRING,
    device_type STRING,
    session_start TIMESTAMP(3),
    session_end TIMESTAMP(3),
    page_count INT,
    total_duration_seconds INT,
    conversion_flag BOOLEAN,
    PRIMARY KEY (session_id) NOT ENFORCED
) WITH (
    'connector' = 'iceberg',
    'catalog-name' = 'lakehouse',
    'database-name' = 'analytics',
    'table-name' = 'user_sessions',
    'write.upsert.enabled' = 'true'
);

-- Pipeline de sessionisation avec fenêtre de session
INSERT INTO user_sessions
SELECT
    session_id,
    FIRST_VALUE(user_id) AS user_id,
    FIRST_VALUE(device_type) AS device_type,
    SESSION_START(event_time, INTERVAL '30' MINUTE) AS session_start,
    SESSION_END(event_time, INTERVAL '30' MINUTE) AS session_end,
    CAST(COUNT(*) AS INT) AS page_count,
    CAST(
        TIMESTAMPDIFF(SECOND,
            SESSION_START(event_time, INTERVAL '30' MINUTE),
            SESSION_END(event_time, INTERVAL '30' MINUTE)
        ) AS INT
    ) AS total_duration_seconds,
    MAX(CASE WHEN action_type = 'purchase' THEN TRUE ELSE FALSE END) AS conversion_flag
FROM raw_clickstream
GROUP BY
    SESSION(event_time, INTERVAL '30' MINUTE),
    session_id;
```

Garanties de Livraison et Exactly-Once

La sémantique de livraison « exactly-once » est cruciale pour un Streaming Lakehouse où la cohérence des données analytiques dépend de l'absence de duplications ou de pertes.

Kafka offre des garanties exactly-once via les transactions et l'idempotence des producteurs. Chaque message est écrit exactement une fois, même en cas de retry.

Flink maintient l'état de manière cohérente via son mécanisme de checkpointing distribué. En cas de défaillance, le traitement reprend depuis le dernier checkpoint avec une cohérence garantie.

Iceberg assure l'atomicité des commits. Chaque commit de fichiers est une opération atomique : soit tous les fichiers sont visibles, soit aucun.

La combinaison de ces garanties permet un pipeline end-to-end exactly-once :

```
// Configuration Flink pour exactly-once avec Iceberg
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

// Activer le checkpointing pour exactly-once
env.enableCheckpointing(60000, CheckpointingMode.EXACTLY_ONCE);
env.getCheckpointConfig().setMinPauseBetweenCheckpoints(30000);
env.getCheckpointConfig().setCheckpointTimeout(120000);
env.getCheckpointConfig().setMaxConcurrentCheckpoints(1);
env.getCheckpointConfig().setExternalizedCheckpointCleanup(
    CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION
);

// Configurer le state backend pour haute disponibilité
env.setStateBackend(new EmbeddedRocksDBStateBackend());
env.getCheckpointConfig().setCheckpointStorage(
    "s3://checkpoints-bucket/flink-checkpoints/"
);

// Configuration Kafka source avec exactly-once
KafkaSource<GenericRecord> kafkaSource = KafkaSource.<GenericRecord>builder()
    .setBootstrapServers("kafka:9092")
    .setTopics("orders.events")
    .setGroupId("flink-iceberg-exactly-once")
    .setStartingOffsets(OffsetsInitializer.committedOffsets(OffsetResetStrategy.EARLIEST))
    .setDeserializer(KafkaRecordDeserializationSchema.of(
        new ConfluentRegistryAvroDeserializationSchema<>(
            GenericRecord.class,
            schemaRegistryUrl
        )
    ))
    .setProperty("isolation.level", "read_committed")  // Important pour exactly-once
    .build();

// Configuration Iceberg sink
FlinkSink.Builder<RowData> sinkBuilder = FlinkSink.forRowData(rowDataStream)
    .tableLoader(tableLoader)
    .overwrite(false)
    .equalityFieldColumns(Arrays.asList("order_id"))  // Clé pour upsert
    .upsert(true);

// Le commit Iceberg est coordonné avec le checkpoint Flink
// garantissant l'exactly-once end-to-end
```

Considérations de Coûts

L'optimisation des coûts dans un Streaming Lakehouse requiert une compréhension des différents postes de dépenses et de leurs leviers.

Coûts Kafka/Confluent : - Débit d'ingestion (par Go ingéré) - Rétention (par Go-heure stocké) - Connecteurs et Flink (par UCU - Confluent Unit Compute)

Coûts Stockage Objet (S3/ADLS/GCS) : - Stockage (par Go/mois) - Requêtes (par millier de requêtes PUT/GET) - Transfert inter-régions (par Go transféré)

Coûts Calcul (Flink, Trino, etc.) : - Instances de traitement (par vCPU-heure) - Mémoire (par Go-heure)

Composant	Coût mensuel estimé (100 To, 1M msg/s)
Confluent Cloud (Kafka + Flink)	15 000 - 25 000 $ CAD
Stockage S3 (ca-central-1)	2 300 $ CAD
Requêtes S3	500 - 1 500 $ CAD
Trino/Dremio (si utilisé)	3 000 - 8 000 $ CAD
Total estimé	**20 800 - 36 800 $ CAD**

Stratégies d'optimisation :

1. **Compaction agressive** : Réduire le nombre de fichiers diminue les coûts de requêtes S3
2. **Tiering de rétention** : Données récentes dans Kafka, historique dans Iceberg
3. **Compression optimale** : Zstandard offre le meilleur ratio compression/performance
4. **Partitionnement intelligent** : Aligner avec les patterns de requête pour maximiser le pruning

Considérations de Performance

L'optimisation des performances dans un Streaming Lakehouse requiert une attention particulière à plusieurs paramètres.

La fréquence de commit détermine le compromis entre latence et efficacité. Des commits fréquents (toutes les minutes) offrent une fraîcheur maximale mais génèrent de nombreux petits fichiers nécessitant une compaction agressive. Des commits moins fréquents (toutes les heures) produisent des fichiers mieux dimensionnés mais augmentent la latence.

Le parallélisme d'écriture influence le débit d'ingestion. Augmenter le nombre de tâches (partitions Kafka, tâches Flink, workers Connect) améliore le débit au prix d'une multiplication des fichiers générés.

La stratégie de partitionnement Iceberg doit aligner avec les patterns de requête. Un partitionnement par date (heure ou jour) convient à la majorité des cas d'usage analytiques.

> **Performance**
> Configuration recommandée pour un Streaming Lakehouse à haut débit : - Intervalle de commit : 5-15 minutes pour les tables analytiques, 1 minute pour les cas d'usage temps réel critiques - Compaction : Activer la compaction automatique avec un seuil de 10-20 fichiers par partition - Taille cible des fichiers : 128-256 Mo pour équilibrer performance de requête et overhead de métadonnées - Write mode : Utiliser Merge-on-Read pour les cas CDC/upsert, Copy-on-Write pour l'append-only

Observabilité du Pipeline Streaming-Lakehouse

La surveillance d'un Streaming Lakehouse nécessite une observabilité de bout en bout, du producteur Kafka jusqu'aux requêtes analytiques sur Iceberg.

```
# Configuration Prometheus pour métriques Streaming Lakehouse
# prometheus.yml
global:
  scrape_interval: 15s
```

```
scrape_configs:
  # Métriques Kafka (JMX Exporter)
  - job_name: 'kafka-brokers'
    static_configs:
      - targets: ['kafka-1:9090', 'kafka-2:9090', 'kafka-3:9090']

  # Métriques Consumer Lag
  - job_name: 'kafka-consumer-lag'
    static_configs:
      - targets: ['kafka-lag-exporter:8080']

  # Métriques Flink
  - job_name: 'flink-jobmanager'
    static_configs:
      - targets: ['flink-jobmanager:9249']

  - job_name: 'flink-taskmanagers'
    static_configs:
      - targets: ['flink-tm-1:9249', 'flink-tm-2:9249']

  # Métriques Iceberg Catalog (Nessie)
  - job_name: 'nessie-catalog'
    static_configs:
      - targets: ['nessie:9091']
```

Les métriques critiques à surveiller incluent :

```
# Règles d'alerte pour Streaming Lakehouse
# alerting_rules.yml
groups:
  - name: streaming_lakehouse_alerts
    rules:
      # Consumer lag critique
      - alert: KafkaConsumerLagCritical
        expr: kafka_consumer_group_lag > 500000
        for: 10m
        labels:
          severity: critical
        annotations:
          summary: "Retard consommateur critique"
          description: "Le groupe {{ $labels.group }} a un retard de {{ $value }} messages
sur {{ $labels.topic }}"

      # Échec de commit Iceberg
      - alert: IcebergCommitFailure
        expr: increase(iceberg_commit_failures_total[5m]) > 0
        for: 1m
        labels:
          severity: critical
        annotations:
          summary: "Échec de commit Iceberg"
          description: "Des commits Iceberg échouent, vérifier les logs"

      # Prolifération de petits fichiers
      - alert: IcebergSmallFilesProliferation
        expr: iceberg_table_files_count > 500 AND iceberg_table_avg_file_size_mb < 50
        for: 2h
        labels:
```

```
          severity: warning
        annotations:
          summary: "Compaction nécessaire"
          description: "La table {{ $labels.table }} contient {{ $value }} fichiers de
taille moyenne {{ $labels.avg_size }}MB"

      # Fraîcheur des données
      - alert: DataFreshnessLag
        expr: (time() - iceberg_table_last_commit_timestamp) > 3600
        for: 15m
        labels:
          severity: warning
        annotations:
          summary: "Données non fraîches"
          description: "La table {{ $labels.table }} n'a pas reçu de commit depuis plus
d'une heure"
```

L'Approche Lakehouse-Native avec Ursa

Une approche émergente mérite mention : les moteurs de streaming « lakehouse-native ». Le projet Ursa, développé par StreamNative et récompensé du Best Industry Paper à VLDB 2025, représente cette nouvelle génération.

Ursa écrit les données directement en format Iceberg ou Delta Lake sur stockage objet, éliminant la distinction entre stockage Kafka et stockage analytique. Chaque topic est simultanément un flux d'événements consommable via l'API Kafka et une table Iceberg interrogeable via SQL. Cette dualité simplifie radicalement l'architecture en éliminant la nécessité de pipelines de synchronisation.

L'architecture d'Ursa utilise un journal d'écriture anticipée (WAL) sur stockage objet pour les données récentes avec latence minimale, tandis qu'un service de compaction en arrière-plan convertit progressivement ces données en fichiers Parquet organisés selon les formats de table ouverts.

Cette approche offre une réduction potentielle des coûts jusqu'à 10× par rapport à un cluster Kafka traditionnel, principalement grâce à l'élimination de la réplication triple et de la duplication streaming/lakehouse.

Considérations pour le Contexte Canadien

Les organisations canadiennes adoptant le Streaming Lakehouse doivent tenir compte de considérations spécifiques liées à la réglementation, à la géographie et à l'écosystème technologique local.

La souveraineté des données requiert que les données sensibles restent dans des régions canadiennes. AWS offre les régions Canada (Montréal) ca-central-1 et Canada (Calgary) ca-west-1, tandis qu'Azure propose Canada Centre (Toronto) et Canada Est (Québec). Google Cloud dispose de la région northamerica-northeast1 (Montréal) et northamerica-northeast2 (Toronto). Confluent Cloud supporte le déploiement dans ces régions, et le stockage Iceberg doit être configuré sur des buckets S3 ou conteneurs Azure situés au Canada.

La conformité Loi 25 (anciennement Projet de loi 64) impose des exigences strictes sur le consentement, la transparence et le droit à l'effacement des données personnelles des résidents québécois. Un Streaming Lakehouse bien conçu facilite la conformité grâce à :

- La traçabilité complète offerte par le journal Kafka (qui a accédé à quoi, quand)
- Le support natif des opérations DELETE par Iceberg pour le droit à l'effacement

- Les capacités de time travel permettant de démontrer l'état des données à un moment donné
- La gouvernance centralisée des schémas documentant la structure des données personnelles

```
# Implémentation du droit à l'effacement (Loi 25, Article 27)
# Pipeline de suppression propagée Kafka -> Iceberg

from dataclasses import dataclass
from typing import List
import json

@dataclass
class ErasureRequest:
    request_id: str
    subject_identifier: str  # Identifiant de la personne concernée
    identifier_type: str     # email, customer_id, etc.
    requested_at: str
    tables_to_process: List[str]

class ErasurePipeline:
    """
    Pipeline de conformité Loi 25 pour le droit à l'effacement.
    Coordonne la suppression entre Kafka et Iceberg.
    """

    def __init__(self, kafka_admin, iceberg_catalog, audit_logger):
        self.kafka_admin = kafka_admin
        self.iceberg_catalog = iceberg_catalog
        self.audit_logger = audit_logger

    def process_erasure_request(self, request: ErasureRequest):
        """
        Traite une demande d'effacement conformément à la Loi 25.
        Délai légal : 30 jours.
        """
        # 1. Journaliser la demande pour audit
        self.audit_logger.log_erasure_request(
            request_id=request.request_id,
            subject=request.subject_identifier,
            timestamp=request.requested_at
        )

        results = {}

        # 2. Supprimer des tables Iceberg
        for table_name in request.tables_to_process:
            table = self.iceberg_catalog.load_table(table_name)

            # Identifier la colonne correspondant à l'identifiant
            id_column = self._get_identifier_column(
                table,
                request.identifier_type
            )

            # Exécuter la suppression
            delete_count = self._execute_delete(
                table=table,
                column=id_column,
                value=request.subject_identifier
            )
```

```
            results[table_name] = {
                'deleted_rows': delete_count,
                'timestamp': datetime.utcnow().isoformat()
            }

            # Journaliser pour audit
            self.audit_logger.log_table_erasure(
                request_id=request.request_id,
                table=table_name,
                rows_deleted=delete_count
            )

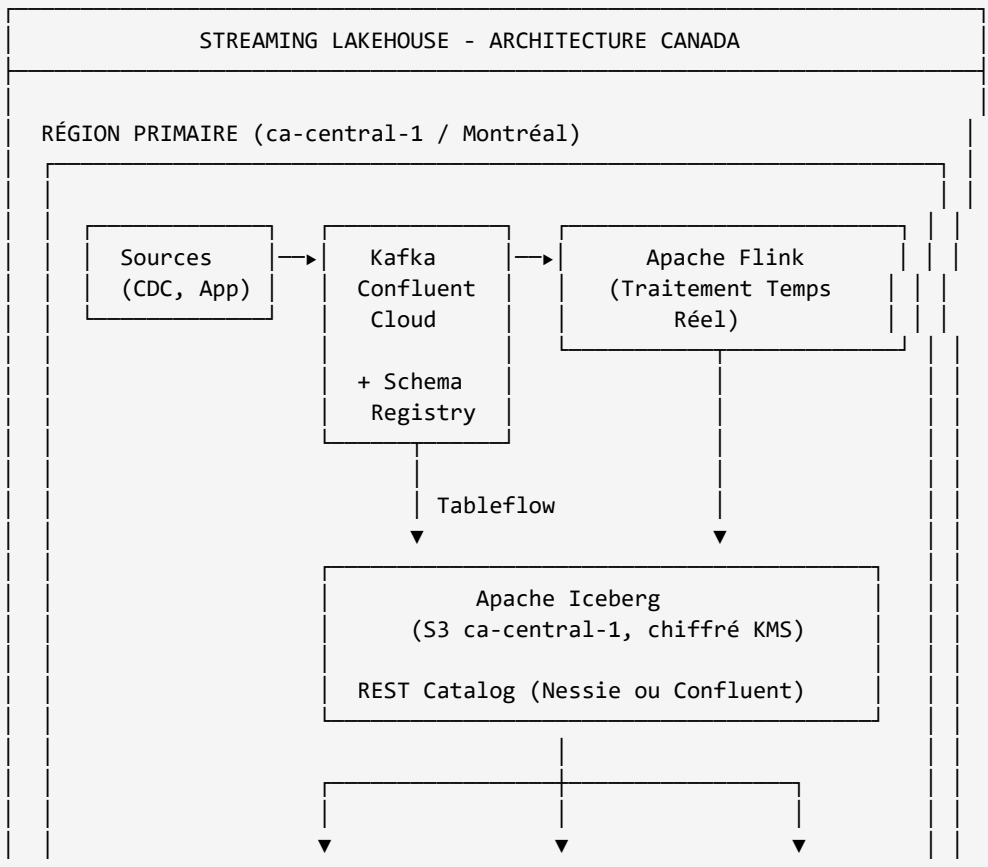
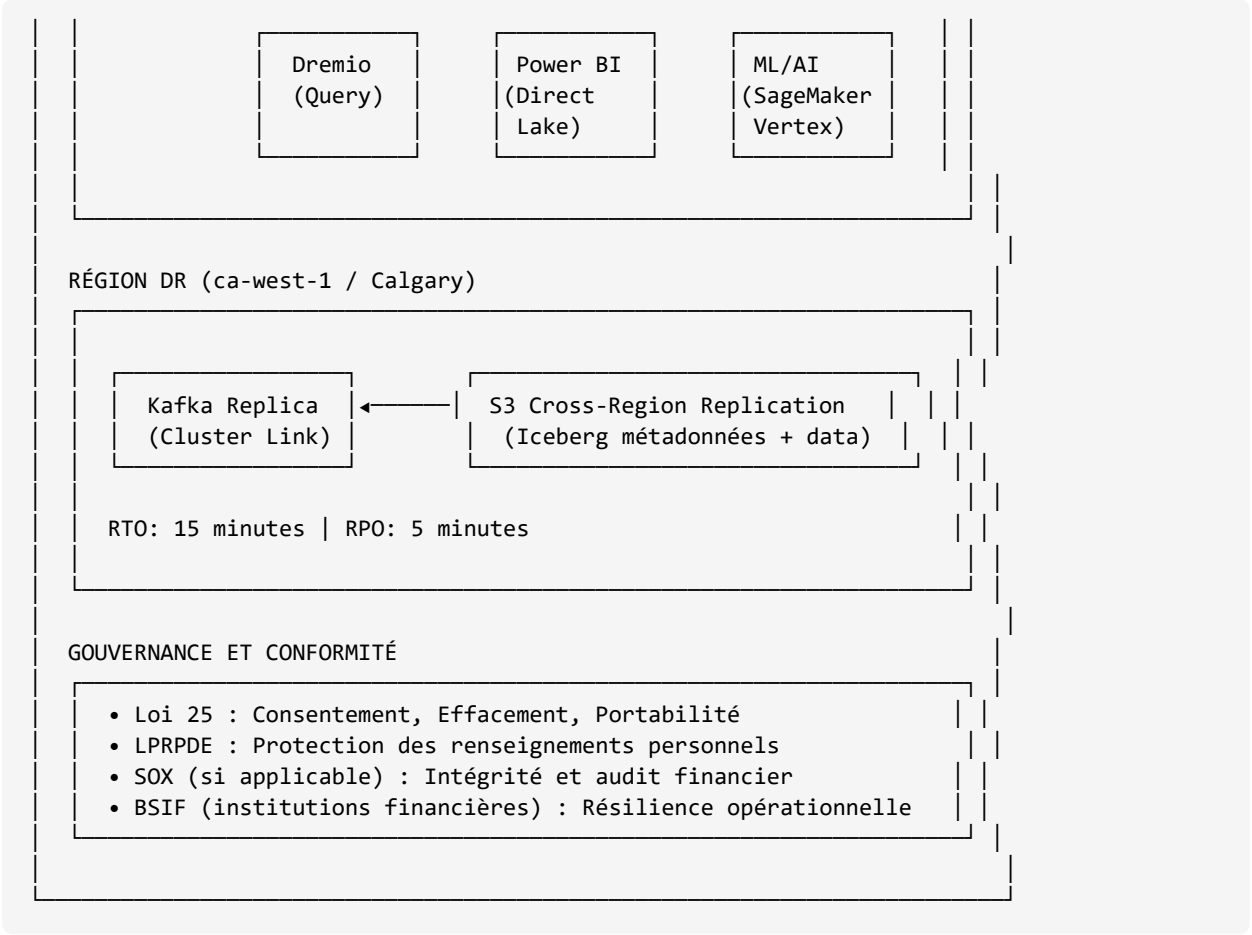
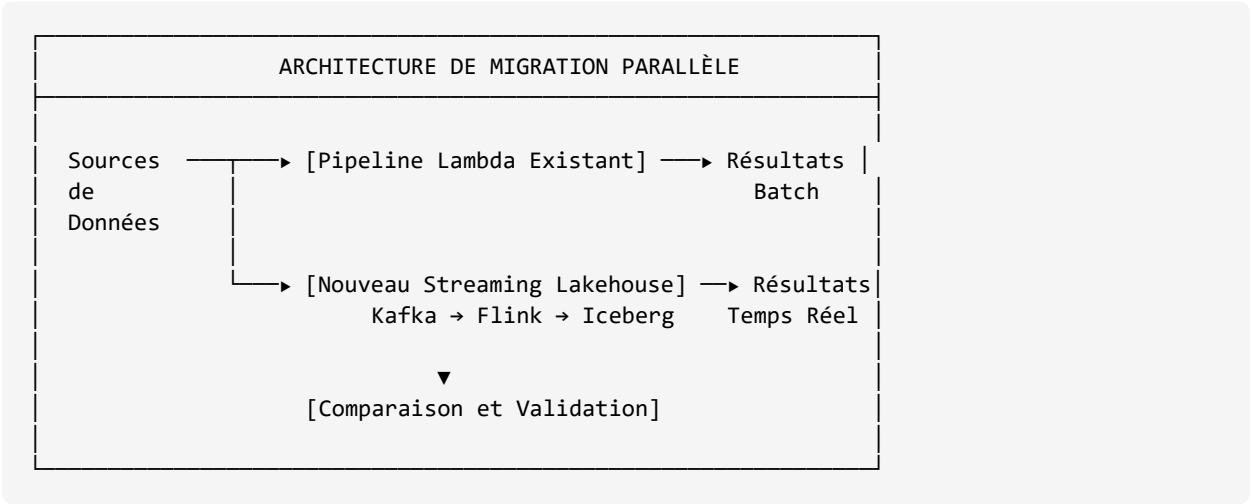
        # 3. Produire un tombstone sur les topics Kafka pertinents
        self._produce_tombstones(
            subject_identifier=request.subject_identifier,
            identifier_type=request.identifier_type
        )

        # 4. Générer le certificat de suppression
        certificate = self._generate_erasure_certificate(
            request=request,
            results=results
        )

        return certificate

    def _execute_delete(self, table, column, value):
        """
        Exécute une suppression SQL sur la table Iceberg.
        Utilise le DELETE natif d'Iceberg (v2).
        """
        # Via Spark SQL ou Flink
        delete_sql = f"""
            DELETE FROM {table.name()}
            WHERE {column} = '{value}'
        """
        return self.spark.sql(delete_sql).count()

    def _produce_tombstones(self, subject_identifier, identifier_type):
        """
        Produit des tombstones Kafka pour les topics contenant
        les données de la personne concernée.
        """
        topics = self._identify_affected_topics(identifier_type)

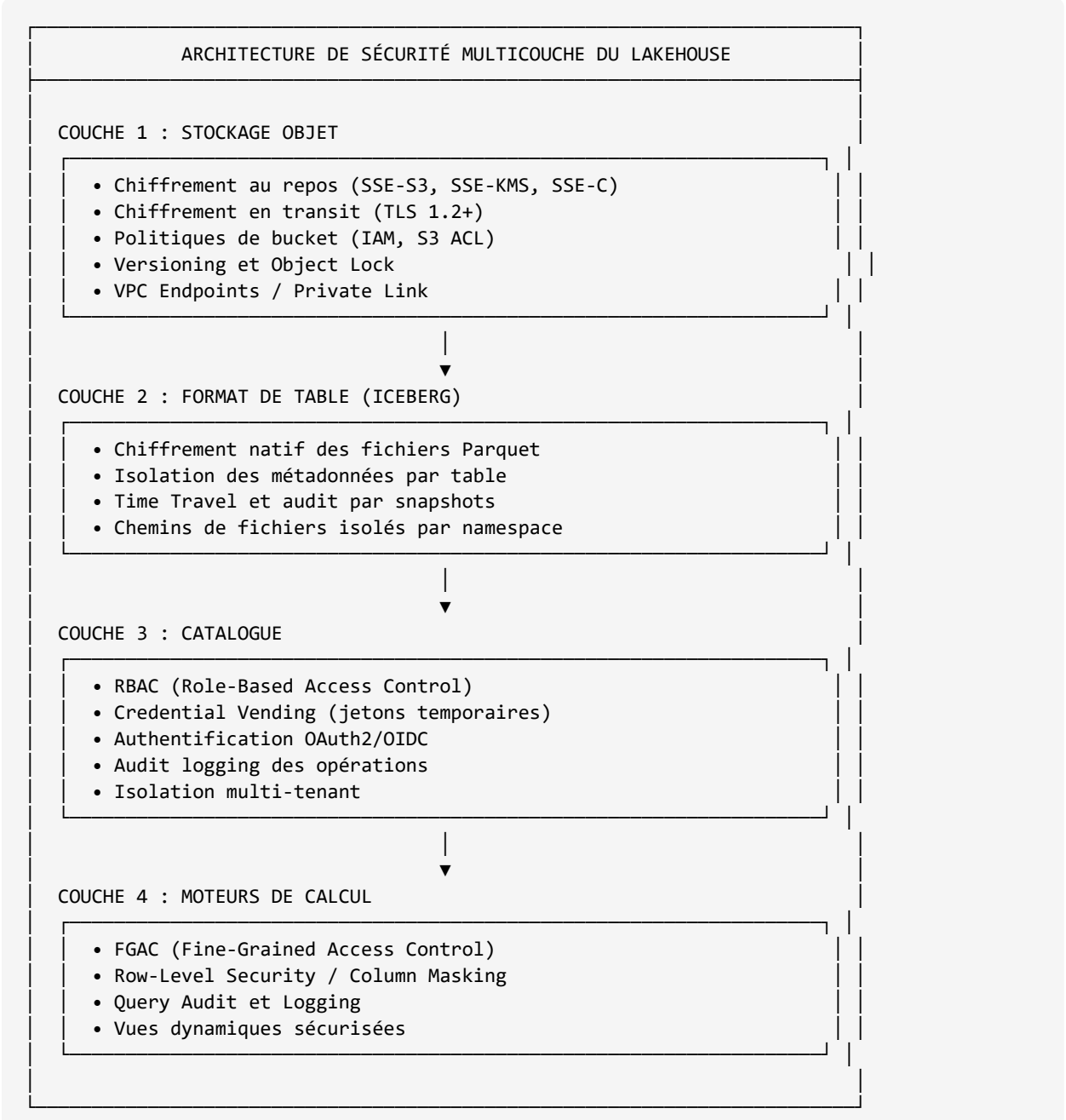
        for topic in topics:
            # Le tombstone (valeur null avec la clé) permet
            # la compaction du log Kafka
            self.kafka_producer.send(
                topic=topic,
                key=subject_identifier,
                value=None  # Tombstone
            )
```

Les coûts de transfert inter-régions peuvent s'accumuler rapidement dans une architecture distribuée. L'optimisation consiste à colocater les composants Kafka, Flink et le stockage Iceberg dans la même région pour minimiser les frais d'egress. Pour les organisations multi-régionales (ex: Québec et Ontario), une

architecture active-passive avec réplication asynchrone peut être plus économique qu'une distribution active-active.

L'écosystème local inclut plusieurs fournisseurs de services gérés avec présence canadienne :

Fournisseur	Service	Régions Canada
Confluent	Confluent Cloud	ca-central-1 (Montréal)
Aiven	Aiven for Apache Kafka	Toronto, Montréal
Amazon	Amazon MSK	ca-central-1, ca-west-1
Microsoft	Azure Event Hubs	Canada Central, Canada East
Starburst	Starburst Galaxy	Canada

Étude de cas : Mouvement Desjardins
Secteur : Services financiers coopératifs
Défi : Unifier les données de 7 millions de membres provenant de multiples systèmes (bancaires, assurances, valeurs mobilières) pour une vue client 360° temps réel, tout en respectant les exigences strictes de l'AMF et de la Loi 25
Solution : Streaming Lakehouse avec Kafka hébergé au Québec (ca-central-1), Apache Flink pour l'agrégation temps réel des profils clients, et Apache Iceberg pour l'historique analytique. CDC Debezium depuis les systèmes legacy AS/400 et Oracle.
Architecture :
- 47 topics Kafka pour les événements temps réel (transactions, connexions, appels)
- 3 clusters Flink pour les différents domaines (bancaire, assurance, investissement)
- REST Catalog Nessie avec gouvernance centralisée
- Stockage Iceberg sur S3 ca-central-1 avec chiffrement KMS
Résultats :
- Latence de mise à jour du profil client réduite de 24 heures à 3 minutes
- Conformité Loi 25 assurée par traçabilité complète et capacité d'effacement
- Réduction de 45 % des coûts par élimination des ETL batch nocturnes
- Temps de réponse aux demandes d'accès de l'AMF réduit de 5 jours à 4 heures

Étude de cas : Bell Canada
Secteur : Télécommunications
Défi : Traiter en temps réel les données de réseau et d'utilisation de 23 millions d'abonnés pour la détection proactive des pannes, l'optimisation de l'expérience client, et la prévention de la fraude
Solution : Architecture Kappa évoluée vers Streaming Lakehouse avec :
- Ingestion temps réel des métriques réseau (15 millions d'événements/seconde en pointe) via Kafka
- Apache Flink pour la détection d'anomalies avec modèles ML embarqués
- Persistance Iceberg pour l'analyse historique et l'entraînement des modèles
- Intégration Power BI via Dremio pour les tableaux de bord opérationnels
Architecture multi-régions :
- Cluster Kafka primaire : Toronto (Canada Central)
- Réplica DR : Montréal (Canada East)
- Latence de réplication : < 100ms
Résultats :

- Temps moyen de détection des incidents réseau réduit de 45 minutes à 90 secondes
- Capacité d'analyse historique sur 2 ans de données réseau (850 To)
- Détection de fraude SIM swap améliorée de 78 %
- Économies de 3,2 M$ CAD/an en coûts d'infrastructure

Étude de cas : Loblaws Digital
Secteur : Commerce de détail alimentaire
Défi : Personnaliser l'expérience client sur les plateformes numériques (PC Optimum, click & collect) en temps réel basé sur le comportement d'achat et les préférences
Solution : Streaming Lakehouse alimentant un moteur de recommandation temps réel :
- Événements de navigation et d'achat capturés via Kafka
- Calcul de features temps réel avec Flink (panier moyen, catégories favorites, sensibilité prix)
- Feature Store matérialisé dans Iceberg pour l'entraînement ML
- Modèles de recommandation servis via API avec latence < 50ms
Résultats :
- Taux de conversion des recommandations augmenté de 23 %
- Valeur moyenne du panier en hausse de 8 %
- Temps d'entraînement des modèles réduit de 4 heures à 45 minutes grâce aux features pré-calculées

Architecture de Référence pour le Canada

L'architecture suivante représente un déploiement de référence pour une organisation canadienne adoptant le Streaming Lakehouse :





Cette architecture assure :

- **Souveraineté des données** : Toutes les données restent dans les régions canadiennes
- **Haute disponibilité** : Réplication entre Montréal et Calgary
- **Conformité** : Traçabilité, audit et capacités d'effacement
- **Performance** : Latence minimale grâce à la colocation des composants
- **Évolutivité** : Scaling indépendant de chaque couche

Feuille de Route pour la Migration

La transition vers un Streaming Lakehouse depuis une architecture Lambda ou des pipelines batch traditionnels requiert une approche méthodique en phases. L'objectif est de minimiser les risques tout en démontrant rapidement la valeur.

Phase 1 : Fondations (Mois 1-3)

Cette phase établit l'infrastructure de base sans perturber les systèmes existants.

Objectifs : - Déployer un cluster Kafka ou Confluent Cloud dans une région canadienne - Établir le Schema Registry avec les premiers schémas Avro - Configurer le stockage Iceberg sur S3/ADLS avec un REST Catalog - Former l'équipe aux concepts fondamentaux

Livrables : - Environnement de développement fonctionnel - Premier topic Kafka alimenté par une source non critique - Première table Iceberg matérialisée (via Tableflow ou Kafka Connect) - Documentation des standards et conventions

Métriques de succès : - Latence de bout en bout < 5 minutes - Zéro perte de données sur un mois de production test

Phase 2 : Cas d'Usage Pilote (Mois 4-6)

Cette phase déploie un premier cas d'usage en production parallèlement aux systèmes existants.

Sélection du cas pilote : Choisir un cas d'usage présentant les caractéristiques suivantes : - Haute valeur métier mais risque limité en cas d'échec - Latence actuelle insatisfaisante (typiquement T+1) - Volume gérable (< 100 Go/jour, < 50 000 événements/seconde) - Équipe métier engagée et disponible pour valider

Exemples de cas pilotes recommandés : - Tableau de bord temps réel pour les indicateurs opérationnels - Détection d'anomalies sur un flux de transactions secondaire - Alimentation d'un Feature Store pour un modèle ML existant

Architecture parallèle : Durant cette phase, le nouveau pipeline Streaming Lakehouse fonctionne en parallèle avec l'existant. Les consommateurs peuvent comparer les résultats des deux systèmes pour valider la cohérence avant basculement.



Livrables : - Pipeline Streaming Lakehouse complet pour le cas pilote - Tableaux de bord d'observabilité - Runbooks opérationnels - Formation des équipes de support

Phase 3 : Extension et Consolidation (Mois 7-12)

Cette phase étend le Streaming Lakehouse à des cas d'usage additionnels et commence le décommissionnement des systèmes legacy.

Objectifs : - Migrer 3 à 5 cas d'usage supplémentaires - Intégrer les sources CDC pour les systèmes transactionnels - Établir la gouvernance centralisée (catalogage, lignage, accès) - Décommissionner les premiers pipelines batch redondants

Considérations critiques : - Maintenir une période de fonctionnement parallèle d'au moins 3 mois avant décommissionnement - Documenter les différences sémantiques entre ancien et nouveau système - Planifier la migration des consommateurs en coordination avec les équipes métier

Phase 4 : Optimisation et Maturité (Année 2)

Cette phase finalise la transition et optimise les coûts et performances.

Objectifs : - Décommissionner l'architecture Lambda complète - Implémenter les patterns avancés (Shift Left, CDC complet) - Optimiser les coûts (compaction, tiering, réservations) - Établir les pratiques MLOps intégrées

Migration
De : Architecture Lambda avec ETL batch nocturnes
Vers : Streaming Lakehouse unifié Kafka-Iceberg
Stratégie : Migration progressive en 4 phases sur 12-18 mois. Fonctionnement parallèle obligatoire avant tout décommissionnement. Budget de contingence de 20 % pour les imprévus techniques.

Résolution des Problèmes Courants

L'exploitation d'un Streaming Lakehouse présente des défis opérationnels spécifiques. Cette section documente les problèmes les plus fréquents et leurs solutions.

Problème : Consumer Lag Croissant

Symptômes : Le retard entre la production et la consommation des messages Kafka augmente progressivement, indiquant que les consommateurs ne parviennent pas à suivre le rythme d'ingestion.

Causes possibles : - Ressources de calcul insuffisantes (tâches Flink sous-dimensionnées) - Backpressure dans le pipeline Flink dû à des opérations coûteuses - Commits Iceberg trop fréquents créant des goulots d'étranglement

Diagnostic :

```
# Vérifier le consumer lag
kafka-consumer-groups.sh --bootstrap-server kafka:9092 \
  --describe --group flink-streaming-lakehouse

# Identifier le backpressure Flink (si disponible)
curl http://flink-jobmanager:8081/jobs/<job-id>/vertices/<vertex-id>/backpressure
```

Solutions : - Augmenter le parallélisme Flink (`SET 'parallelism.default' = '16';`) - Optimiser les opérations coûteuses (éviter les aggregations non-bornées) - Réduire la fréquence de checkpoint si elle est excessive - Augmenter l'intervalle de commit Iceberg (5-15 minutes au lieu de 1 minute)

Problème : Prolifération de Petits Fichiers

Symptômes : Les performances de requête sur les tables Iceberg se dégradent progressivement. Le nombre de fichiers par partition dépasse plusieurs centaines.

Causes possibles : - Commits trop fréquents générant de nombreux fichiers sous-dimensionnés - Parallélisme d'écriture élevé combiné à un faible volume - Compaction désactivée ou mal configurée

Diagnostic :

```
-- Analyser la distribution des fichiers
SELECT
    partition,
    COUNT(*) as file_count,
    SUM(file_size_in_bytes) / 1024 / 1024 as total_size_mb,
    AVG(file_size_in_bytes) / 1024 / 1024 as avg_file_size_mb
FROM prod.lakehouse.orders.files
GROUP BY partition
HAVING file_count > 100
ORDER BY file_count DESC;
```

Solutions : - Activer la compaction automatique avec des seuils appropriés - Augmenter l'intervalle entre les commits - Réduire le parallélisme d'écriture si le volume est faible - Exécuter une compaction manuelle pour les tables problématiques

```
-- Compaction manuelle (Spark)
CALL prod.system.rewrite_data_files(
    table => 'lakehouse.orders',
    options => map('target-file-size-bytes', '268435456')  -- 256 MB
);
```

Problème : Échecs de Checkpoint Flink

Symptômes : Les checkpoints Flink échouent régulièrement, provoquant des retours en arrière et des duplications potentielles.

Causes possibles : - Timeout de checkpoint trop court pour le volume d'état - State backend mal configuré ou saturé - Problèmes de connectivité avec le stockage des checkpoints

Solutions : - Augmenter le timeout de checkpoint (`execution.checkpointing.timeout: 10min`) - Activer les checkpoints incrémentaux pour RocksDB - Vérifier la latence vers le stockage S3/ADLS - Réduire la taille de l'état en utilisant des TTL sur les états non nécessaires

```
// Configuration du TTL pour les états Flink
StateTtlConfig ttlConfig = StateTtlConfig
    .newBuilder(Time.hours(24))  // Expiration après 24h d'inactivité
    .setUpdateType(StateTtlConfig.UpdateType.OnReadAndWrite)
    .setStateVisibility(StateTtlConfig.StateVisibility.NeverReturnExpired)
    .build();

ValueStateDescriptor<UserState> stateDescriptor =
    new ValueStateDescriptor<>("user-state", UserState.class);
stateDescriptor.enableTimeToLive(ttlConfig);
```

Problème : Divergence de Schéma entre Kafka et Iceberg

Symptômes : Les écritures vers Iceberg échouent avec des erreurs de schéma incompatible, bien que le Schema Registry accepte les évolutions.

Causes possibles : - Évolution de schéma dans le Registry non propagée vers Iceberg - Types incompatibles entre Avro et Iceberg - Champs obligatoires ajoutés sans valeur par défaut

Solutions : - Vérifier que l'évolution de schéma respecte les règles de compatibilité Iceberg - Utiliser FULL ou FULL_TRANSITIVE pour une compatibilité stricte - Toujours définir des valeurs par défaut pour les nouveaux champs - Recréer le connecteur après évolution majeure si nécessaire

Problème : Latence de Requête Élevée

Symptômes : Les requêtes analytiques sur les tables Iceberg prennent plusieurs minutes au lieu des secondes attendues.

Causes possibles : - Partitionnement inadapté aux patterns de requête - Absence de pruning sur les prédicats de filtre - Trop de fichiers nécessitant un scan complet - Métriques de colonnes non maintenues

Solutions : - Analyser le plan d'exécution pour identifier les scans complets - Revoir la stratégie de partitionnement - Activer les statistiques de colonnes - Considérer le tri (clustering) sur les colonnes de filtre fréquentes

```
-- Ajouter le tri sur les colonnes fréquemment filtrées (Spark)
ALTER TABLE prod.lakehouse.orders
WRITE ORDERED BY customer_id, order_date;

-- Réécrire avec le nouveau tri
CALL prod.system.rewrite_data_files(
    table => 'lakehouse.orders',
    strategy => 'sort'
);
```

Gestion des Données Tardives

Dans un système de streaming, les données arrivent parfois avec un retard significatif par rapport à leur timestamp d'événement. Cette situation, fréquente dans les architectures IoT ou mobiles, nécessite une gestion explicite pour maintenir l'intégrité analytique.

Apache Flink utilise le concept de watermark pour délimiter la frontière entre les données attendues et les données tardives. Le watermark représente une assertion que tous les événements avec un timestamp inférieur sont arrivés (ou devraient être considérés comme tels).

```
-- Configuration du watermark avec tolérance pour les retards
CREATE TABLE mobile_events (
    event_id STRING,
    user_id STRING,
    event_type STRING,
    event_time TIMESTAMP(3),
    device_info STRING,
    -- Watermark avec tolérance de 5 minutes pour les événements mobiles
    WATERMARK FOR event_time AS event_time - INTERVAL '5' MINUTE
) WITH (
    'connector' = 'kafka',
    'topic' = 'mobile.events',
    'properties.bootstrap.servers' = 'kafka:9092',
    'format' = 'avro-confluent',
    'avro-confluent.url' = 'http://schema-registry:8081'
);

-- Stratégie pour les données extrêmement tardives
-- Option 1 : Table séparée pour les données tardives
CREATE TABLE late_mobile_events (
    event_id STRING,
    user_id STRING,
    event_type STRING,
    event_time TIMESTAMP(3),
    arrival_time TIMESTAMP(3) DEFAULT PROCTIME(),
    PRIMARY KEY (event_id) NOT ENFORCED
) WITH (
    'connector' = 'iceberg',
    'catalog-name' = 'lakehouse',
    'database-name' = 'staging',
    'table-name' = 'late_mobile_events'
);

-- Pipeline avec gestion des données tardives
INSERT INTO late_mobile_events
SELECT
```

```
    event_id,
    user_id,
    event_type,
    event_time,
    PROCTIME() as arrival_time
FROM mobile_events
WHERE event_time < CURRENT_WATERMARK(event_time) - INTERVAL '5' MINUTE;
```

Les données tardives capturées séparément peuvent ensuite être réconciliées via des processus batch périodiques qui mettent à jour les agrégations Iceberg. Le support des opérations MERGE d'Iceberg v2 facilite cette réconciliation.

```
-- Réconciliation batch des données tardives (exécuté quotidiennement)
MERGE INTO analytics.daily_metrics AS target
USING (
    SELECT
        DATE(event_time) as metric_date,
        user_id,
        COUNT(*) as late_event_count,
        SUM(CASE WHEN event_type = 'purchase' THEN 1 ELSE 0 END) as late_purchases
    FROM staging.late_mobile_events
    WHERE arrival_time >= CURRENT_DATE - INTERVAL 1 DAY
    GROUP BY DATE(event_time), user_id
) AS source
ON target.metric_date = source.metric_date
   AND target.user_id = source.user_id
WHEN MATCHED THEN UPDATE SET
    event_count = target.event_count + source.late_event_count,
    purchase_count = target.purchase_count + source.late_purchases,
    last_updated = CURRENT_TIMESTAMP
WHEN NOT MATCHED THEN INSERT (metric_date, user_id, event_count, purchase_count,
last_updated)
VALUES (source.metric_date, source.user_id, source.late_event_count,
source.late_purchases, CURRENT_TIMESTAMP);
```

Cette approche hybride combine la faible latence du streaming pour la majorité des données avec la complétude du batch pour les données tardives, offrant le meilleur des deux mondes.

IV.12.3 Résumé

Ce chapitre a retracé l'évolution des architectures de données depuis l'approche Lambda hybride jusqu'au Streaming Lakehouse moderne unifié. Les points essentiels à retenir sont structurés ci-dessous.

L'Évolution Architecturale

L'architecture Lambda (2011) a posé les fondations en reconnaissant le besoin de combiner traitement batch et temps réel. Cependant, sa dualité intrinsèque engendre une duplication de code, une divergence des résultats et une complexité opérationnelle qui deviennent rapidement insurmontables à l'échelle de l'entreprise.

L'architecture Kappa (2014) a simplifié radicalement l'approche en traitant toutes les données comme un flux continu, avec le journal Kafka comme source de vérité unique. Cette unification élimine la duplication mais limite l'accès SQL et peut s'avérer coûteuse pour le stockage long terme.

Le Data Lakehouse (2017+) a apporté les garanties transactionnelles et l'interrogation SQL aux lacs de données via des formats comme Apache Iceberg, mais reste fondamentalement orienté batch.

Le Streaming Lakehouse (2023+) synthétise ces approches en combinant Kafka pour les données en mouvement et Iceberg pour les données au repos, avec une matérialisation automatique entre les deux couches. Cette architecture unifie enfin streaming et analytique dans une plateforme cohérente.

Le Rôle Central de Kafka et Confluent

Apache Kafka s'impose comme la colonne vertébrale événementielle du Streaming Lakehouse, offrant le découplage, la persistance et l'ordonnancement nécessaires à une architecture de données fiable.

Tableflow de Confluent révolutionne l'intégration Kafka-Iceberg en éliminant les pipelines ETL traditionnels. La matérialisation automatique des topics en tables Iceberg, avec gestion native de l'évolution de schéma et de la compaction, réduit drastiquement la complexité opérationnelle.

Le Schema Registry assure la gouvernance des contrats de données entre producteurs et consommateurs, garantissant la cohérence structurelle de bout en bout.

Patterns d'Implémentation

Trois patterns principaux permettent l'alimentation du Lakehouse depuis Kafka :

1. **Tableflow** pour une matérialisation simple sans code
2. **Kafka Connect** avec Iceberg Sink pour une intégration flexible avec contrôle fin
3. **Apache Flink** pour les transformations complexes et le routage dynamique

Le choix du pattern dépend des exigences de transformation, de l'expertise existante et du niveau de contrôle souhaité.

Considérations Opérationnelles

L'observabilité de bout en bout est essentielle, couvrant les métriques Kafka (consumer lag, throughput), Flink (checkpointing, backpressure) et Iceberg (commits, file count, compaction).

La fréquence de commit constitue le principal levier de compromis entre latence et efficacité du stockage. Une compaction automatique bien configurée maintient les performances de requête.

Recommandations Stratégiques

Pour les organisations qui entreprennent la transition vers le Streaming Lakehouse, les recommandations suivantes guident l'adoption :

1. **Commencer par les cas d'usage à haute valeur** où la latence actuelle constitue un frein tangible (détection de fraude, pricing dynamique, tableaux de bord temps réel)
2. **Adopter le pattern Shift Left** en déplaçant la gouvernance et la qualité des données vers la couche streaming plutôt que dans des processus ETL tardifs
3. **Standardiser sur Apache Iceberg** comme format de table pour maximiser l'interopérabilité entre les moteurs de consommation

4. **Investir dans l'observabilité** de bout en bout, du producteur Kafka jusqu'aux requêtes analytiques sur Iceberg
5. **Considérer Tableflow** pour les nouveaux cas d'usage afin de minimiser le code et la maintenance opérationnelle
6. **Planifier la migration incrémentale** depuis Lambda ou les architectures batch existantes, en maintenant les systèmes en parallèle pendant la transition
7. **Assurer la souveraineté des données** en déployant tous les composants (Kafka, Flink, stockage Iceberg) dans des régions canadiennes pour la conformité Loi 25 et LPRPDE

Le Streaming Lakehouse représente l'état de l'art pour les architectures de données modernes. En unifiant les données en mouvement et au repos sous une gouvernance commune, il permet aux organisations de répondre aux exigences de temps réel sans sacrifier la complétude historique ni la rigueur analytique. Pour les organisations canadiennes, cette architecture offre en outre un cadre robuste pour la conformité réglementaire grâce à sa traçabilité inhérente et son support natif des opérations d'effacement.

Références

1. Confluent (2025). *Introducing Tableflow: Unifying Streaming and Analytics*. Documentation technique Confluent.
2. Apache Flink (2025). *From Stream to Lakehouse: Kafka Ingestion with the Flink Dynamic Iceberg Sink*. Blog officiel Apache Flink.
3. Waehner, K. (2025). *The Rise of Kappa Architecture in the Era of Agentic AI and Data Streaming*. Blog technique.
4. Waehner, K. (2025). *Data Streaming Meets Lakehouse: Apache Iceberg for Unified Real-Time and Batch Analytics*. Présentation Open Source Data Summit.
5. Ververica (2025). *From Kappa Architecture to Streamhouse: Making the Lakehouse Real-Time*. Documentation technique.
6. StreamNative (2025). *Ursa Wins VLDB 2025 Best Industry Paper: The First Lakehouse-Native Streaming Engine for Kafka*. Communiqué de presse.
7. Apache Iceberg (2025). *Flink Getting Started*. Documentation officielle.
8. Merced, A. (2024). *2025 Guide to Architecting an Iceberg Lakehouse*. Medium.
9. Kreps, J. (2014). *Questioning the Lambda Architecture*. Blog O'Reilly.
10. Marz, N. (2011). *How to beat the CAP theorem*. Blog personnel.

Chapitre IV.13 - SÉCURITÉ, GOUVERNANCE ET CONFORMITÉ DU LAKEHOUSE

Introduction

La démocratisation des données constitue l'une des promesses les plus séduisantes du Data Lakehouse. En centralisant les données organisationnelles dans un format ouvert et accessible par une multitude de moteurs de requête, l'architecture Iceberg permet à chaque analyste, scientifique de données et application d'accéder aux informations dont ils ont besoin. Cependant, cette accessibilité accrue s'accompagne d'une responsabilité proportionnelle : protéger les données sensibles, contrôler les accès, et démontrer la conformité aux cadres réglementaires de plus en plus stricts.

Pour les organisations canadiennes, ces enjeux revêtent une importance particulière. La Loi 25 au Québec, entrée pleinement en vigueur en septembre 2024, impose des exigences strictes en matière de consentement explicite, de droit à l'effacement et de portabilité des données. La LPRPDE (Loi sur la protection des renseignements personnels et les documents électroniques) au niveau fédéral, actuellement en cours de modernisation, établit les principes fondamentaux de protection de la vie privée. Dans les secteurs réglementés tels que les services financiers, les directives du BSIF ajoutent des exigences supplémentaires de résilience opérationnelle et de gestion des risques technologiques.

Ce chapitre présente une approche holistique de la sécurité et de la gouvernance du Lakehouse. Nous examinerons d'abord l'architecture de sécurité multicouche qui protège les données à chaque niveau, du stockage objet au catalogue. Puis nous explorerons les mécanismes de contrôle d'accès, des permissions grossières au niveau des tables jusqu'au masquage dynamique au niveau des colonnes. Ensuite, nous aborderons la gouvernance des données à travers le lignage, la qualité et l'observabilité. Enfin, nous présenterons les stratégies de conformité spécifiques au contexte canadien, illustrées par des études de cas d'organisations ayant réussi cette transformation.

IV.13.1 Architecture de Sécurité Multicouche

Le Défi de la Sécurité Distribuée

L'architecture du Data Lakehouse introduit des défis de sécurité uniques par rapport aux systèmes monolithiques traditionnels. Dans un entrepôt de données classique, un seul système contrôle l'accès aux données, simplifiant la gestion des permissions. Dans un Lakehouse, les données résident sur un stockage objet (S3, ADLS, GCS), les métadonnées sont gérées par un catalogue (REST Catalog, Glue, Polaris), et de multiples moteurs de requête (Spark, Flink, Trino, Dremio) peuvent accéder aux mêmes tables. Cette distribution crée des vecteurs d'attaque potentiels à chaque couche.

La spécification Apache Iceberg elle-même ne définit pas de mécanismes de gouvernance ou de contrôle d'accès. Ce choix architectural délibéré permet la flexibilité et l'interopérabilité, mais transfère la responsabilité de la sécurité aux couches adjacentes de l'écosystème. Une stratégie de sécurité efficace doit donc couvrir simultanément le stockage, le catalogue et les moteurs de calcul.



Sécurité du Stockage Objet

La première ligne de défense réside dans le stockage objet qui héberge les fichiers de données et de métadonnées Iceberg. Cette couche doit assurer la confidentialité, l'intégrité et la disponibilité des données au repos.

Le chiffrement au repos constitue la protection fondamentale. AWS S3 offre trois options de chiffrement côté serveur. SSE-S3 utilise des clés gérées par AWS, offrant simplicité et absence de coût supplémentaire. SSE-KMS utilise AWS Key Management Service, permettant le contrôle des clés, l'audit de leur utilisation

et la rotation automatique. SSE-C (clés fournies par le client) offre le contrôle maximal mais nécessite une gestion des clés par l'organisation.

Pour les organisations canadiennes soumises à des exigences réglementaires strictes, SSE-KMS avec des clés CMK (Customer Master Key) hébergées dans une région canadienne représente généralement le meilleur compromis entre sécurité et opérabilité.

```
# Configuration Terraform pour un bucket S3 sécurisé hébergeant Iceberg
# Région canadienne avec chiffrement KMS et versioning

resource "aws_kms_key" "lakehouse_key" {
  description             = "Clé KMS pour le chiffrement du Lakehouse"
  deletion_window_in_days = 30
  enable_key_rotation     = true

  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Sid    = "Enable IAM User Permissions"
        Effect = "Allow"
        Principal = {
          AWS = "arn:aws:iam::${data.aws_caller_identity.current.account_id}:root"
        }
        Action   = "kms:*"
        Resource = "*"
      },
      {
        Sid    = "Allow Lakehouse Services"
        Effect = "Allow"
        Principal = {
          Service = ["s3.amazonaws.com", "glue.amazonaws.com"]
        }
        Action = [
          "kms:Encrypt",
          "kms:Decrypt",
          "kms:GenerateDataKey*"
        ]
        Resource = "*"
      }
    ]
  })

  tags = {
    Environment = "production"
    DataClass   = "confidential"
    Compliance  = "loi25-lprpde"
  }
}

resource "aws_s3_bucket" "lakehouse" {
  bucket = "entreprise-lakehouse-ca-central-1"

  tags = {
    Environment = "production"
    DataClass   = "confidential"
  }
}
```

```
resource "aws_s3_bucket_versioning" "lakehouse" {
  bucket = aws_s3_bucket.lakehouse.id
  versioning_configuration {
    status = "Enabled"
  }
}

resource "aws_s3_bucket_server_side_encryption_configuration" "lakehouse" {
  bucket = aws_s3_bucket.lakehouse.id

  rule {
    apply_server_side_encryption_by_default {
      kms_master_key_id = aws_kms_key.lakehouse_key.arn
      sse_algorithm     = "aws:kms"
    }
    bucket_key_enabled = true  # Réduit les coûts KMS
  }
}

resource "aws_s3_bucket_public_access_block" "lakehouse" {
  bucket = aws_s3_bucket.lakehouse.id

  block_public_acls       = true
  block_public_policy     = true
  ignore_public_acls      = true
  restrict_public_buckets = true
}

# Politique de bucket restrictive
resource "aws_s3_bucket_policy" "lakehouse" {
  bucket = aws_s3_bucket.lakehouse.id

  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Sid       = "EnforceHTTPS"
        Effect    = "Deny"
        Principal = "*"
        Action    = "s3:*"
        Resource = [
          aws_s3_bucket.lakehouse.arn,
          "${aws_s3_bucket.lakehouse.arn}/*"
        ]
        Condition = {
          Bool = {
            "aws:SecureTransport" = "false"
          }
        }
      },
      {
        Sid       = "RestrictToVPCEndpoint"
        Effect    = "Deny"
        Principal = "*"
        Action    = "s3:*"
        Resource = [
          aws_s3_bucket.lakehouse.arn,
          "${aws_s3_bucket.lakehouse.arn}/*"
```

```
        ]
        Condition = {
          StringNotEquals = {
            "aws:SourceVpce" = var.vpc_endpoint_id
          }
        }
      }
    ]
  })
}
```

L'isolation réseau via VPC Endpoints (AWS) ou Private Link (Azure, GCP) garantit que le trafic vers le stockage ne transite jamais par l'internet public. Cette configuration est particulièrement importante pour les données sensibles et constitue souvent une exigence explicite des politiques de sécurité d'entreprise.

Le Chiffrement Natif Iceberg

Au-delà du chiffrement au niveau du stockage, Apache Iceberg supporte le chiffrement natif des fichiers Parquet. Ce mécanisme permet un contrôle plus granulaire, où différentes colonnes peuvent être chiffrées avec différentes clés.

Le chiffrement Iceberg utilise une architecture à deux niveaux de clés. Les clés de chiffrement de données (DEK - Data Encryption Keys) chiffrent les données elles-mêmes. Ces DEK sont ensuite chiffrées par des clés maîtresses (KEK - Key Encryption Keys) stockées dans un système de gestion de clés externe comme AWS KMS ou HashiCorp Vault.

```
// Configuration du chiffrement Iceberg avec Spark
SparkConf conf = new SparkConf()
    .setAppName("Lakehouse-Encrypted")
    // Configuration du gestionnaire de clés
    .set("spark.sql.catalog.lakehouse.io-impl",
         "org.apache.iceberg.aws.s3.S3FileIO")
    .set("spark.sql.catalog.lakehouse.encryption.manager",
         "org.apache.iceberg.encryption.StandardEncryptionManager")
    .set("spark.sql.catalog.lakehouse.encryption.kms",
         "org.apache.iceberg.aws.kms.AwsKmsClient")
    .set("spark.sql.catalog.lakehouse.encryption.kms.key-id",
         "arn:aws:kms:ca-central-1:123456789:key/abcd-1234");

// Création d'une table avec chiffrement par colonne
spark.sql("""
    CREATE TABLE lakehouse.clients.donnees_personnelles (
        client_id STRING,
        nom STRING,
        prenom STRING,
        nas STRING,           -- Numéro d'assurance sociale (chiffré)
        date_naissance DATE,  -- (chiffré)
        courriel STRING,
        telephone STRING
    )
    USING iceberg
    TBLPROPERTIES (
        'encryption.column.nas' = 'key1',
        'encryption.column.date_naissance' = 'key1',
        'encryption.key.key1' = 'arn:aws:kms:ca-central-1:123456789:key/nas-key'
```

```
    )
""");
```

Cette approche offre plusieurs avantages pour la conformité réglementaire. Les colonnes contenant des données sensibles (numéro d'assurance sociale, informations médicales, données financières) peuvent être protégées indépendamment. L'accès aux clés de déchiffrement peut être contrôlé séparément de l'accès aux tables, permettant une séparation des responsabilités.

Sécurité au Niveau du Catalogue

Le catalogue représente le point de contrôle centralisé pour la gouvernance du Lakehouse. Les catalogues modernes comme Apache Polaris, Unity Catalog ou Nessie offrent des capacités de sécurité avancées qui vont bien au-delà du simple stockage de métadonnées.

Apache Polaris, en incubation à la fondation Apache, émerge comme le standard de facto pour les catalogues Iceberg. Lancé en 2024 par Snowflake et donné à la communauté open source, Polaris implémente intégralement l'API REST Iceberg et ajoute des capacités de gouvernance essentielles.

Le modèle RBAC de Polaris distingue deux types de rôles. Les **Principal Roles** sont attribués aux utilisateurs ou aux applications (service principals) et définissent leur identité dans le système. Les **Catalog Roles** définissent les permissions sur les objets (catalogues, namespaces, tables, vues) et sont attribués aux Principal Roles.

```
# Configuration d'Apache Polaris avec RBAC
# Création des rôles et attribution des permissions

import requests
from typing import Dict, Any

class PolarisAdminClient:
    def __init__(self, base_url: str, admin_token: str):
        self.base_url = base_url
        self.headers = {
            "Authorization": f"Bearer {admin_token}",
            "Content-Type": "application/json"
        }

    def create_catalog_role(self, catalog: str, role_name: str,
                            privileges: list) -> Dict[str, Any]:
        """Crée un rôle de catalogue avec des privilèges spécifiques."""
        payload = {
            "name": role_name,
            "properties": {}
        }

        response = requests.post(
            f"{self.base_url}/api/v1/catalogs/{catalog}/catalog-roles",
            headers=self.headers,
            json=payload
        )
        response.raise_for_status()

        # Attribution des privilèges au rôle
        for privilege in privileges:
            self._grant_privilege(catalog, role_name, privilege)
```

```
        return response.json()

    def _grant_privilege(self, catalog: str, role_name: str,
                         privilege: Dict) -> None:
        """Attribue un privilège à un rôle de catalogue."""
        response = requests.put(
            f"{self.base_url}/api/v1/catalogs/{catalog}/catalog-roles/"
            f"{role_name}/grants",
            headers=self.headers,
            json=privilege
        )
        response.raise_for_status()

    def create_principal_role(self, role_name: str) -> Dict[str, Any]:
        """Crée un rôle principal."""
        payload = {
            "name": role_name,
            "properties": {}
        }

        response = requests.post(
            f"{self.base_url}/api/v1/principal-roles",
            headers=self.headers,
            json=payload
        )
        response.raise_for_status()
        return response.json()

    def assign_catalog_role_to_principal(self, principal_role: str,
                                         catalog: str,
                                         catalog_role: str) -> None:
        """Attribue un rôle de catalogue à un rôle principal."""
        response = requests.put(
            f"{self.base_url}/api/v1/principal-roles/{principal_role}/"
            f"catalog-roles/{catalog}/{catalog_role}",
            headers=self.headers
        )
        response.raise_for_status()


# Exemple d'utilisation : Configuration RBAC pour une équipe analytique
admin = PolarisAdminClient(
    base_url="https://polaris.entreprise.ca",
    admin_token="admin_token_secret"
)

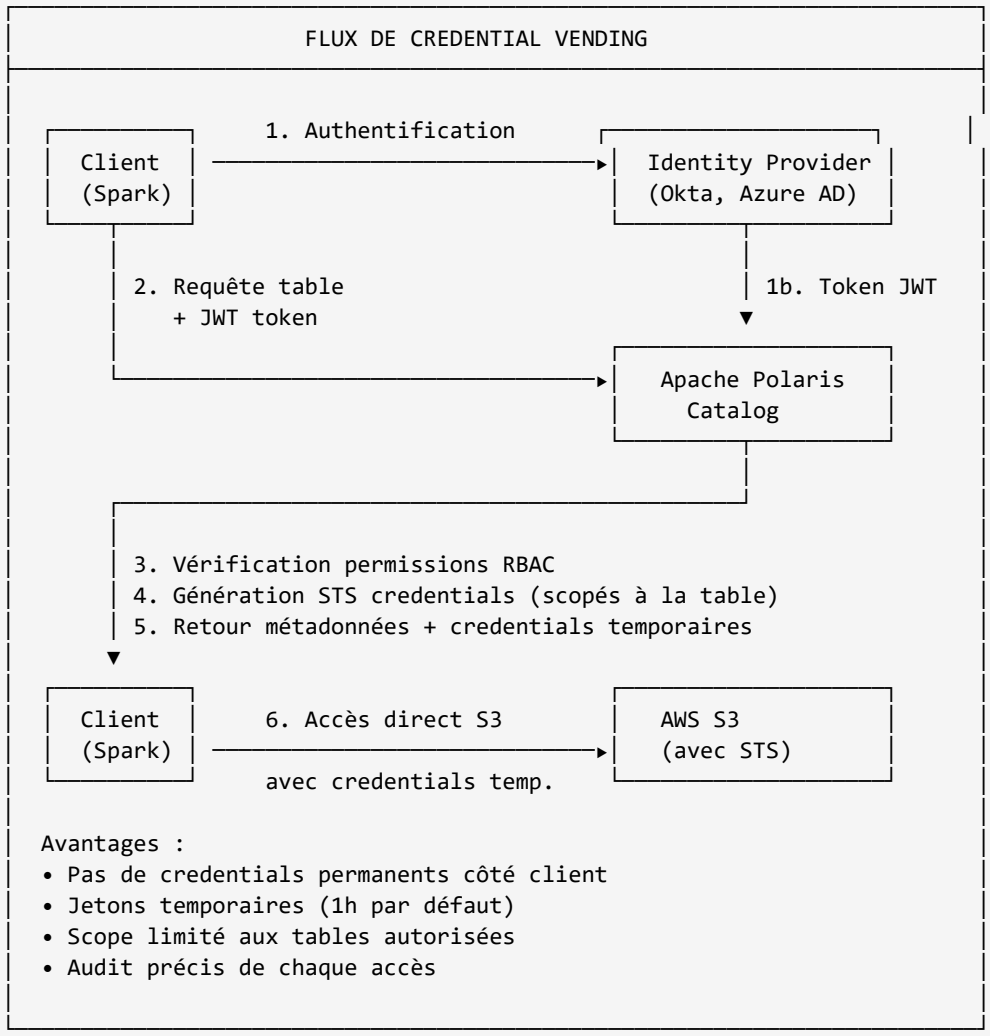
# Créer les rôles de catalogue avec différents niveaux d'accès
admin.create_catalog_role(
    catalog="production",
    role_name="analyst_readonly",
    privileges=[
        {
            "type": "TABLE_READ",
            "namespace": "ventes",
            "table": "*"
        },
        {
            "type": "TABLE_READ",
```

```
            "namespace": "marketing",
            "table": "*"
        }
    ]
)

admin.create_catalog_role(
    catalog="production",
    role_name="data_engineer",
    privileges=[
        {
            "type": "TABLE_FULL",
            "namespace": "*",
            "table": "*"
        },
        {
            "type": "NAMESPACE_CREATE",
            "namespace": "*"
        }
    ]
)

# Restreindre l'accès aux données sensibles
admin.create_catalog_role(
    catalog="production",
    role_name="pii_access",
    privileges=[
        {
            "type": "TABLE_READ",
            "namespace": "clients",
            "table": "donnees_personnelles"
        }
    ]
)

# Créer les rôles principaux et les attributions
admin.create_principal_role("equipe_analytique")
admin.create_principal_role("equipe_ingenierie")
admin.create_principal_role("equipe_conformite")

# Attribuer les rôles de catalogue aux rôles principaux
admin.assign_catalog_role_to_principal(
    principal_role="equipe_analytique",
    catalog="production",
    catalog_role="analyst_readonly"
)

admin.assign_catalog_role_to_principal(
    principal_role="equipe_conformite",
    catalog="production",
    catalog_role="pii_access"
)
```

Le Credential Vending représente une innovation majeure des catalogues modernes. Plutôt que de configurer des identifiants statiques (access keys) sur chaque moteur de calcul, le catalogue émet des jetons temporaires (STS tokens) à la demande. Ces jetons ont une durée de vie limitée (typiquement 1 heure) et sont scopés aux tables spécifiques que l'utilisateur est autorisé à accéder.

Ce mécanisme offre plusieurs avantages de sécurité. Il élimine la nécessité de distribuer et gérer des identifiants permanents. En cas de compromission, les jetons expirent automatiquement. L'audit devient plus précis car chaque accès est associé à un jeton spécifique. Les politiques d'accès sont appliquées de manière cohérente quel que soit le moteur de calcul utilisé.



Project Nessie : Git pour les Données

Project Nessie offre une approche alternative à la gouvernance du catalogue, inspirée des concepts de contrôle de version Git. Développé par Dremio et donné à la communauté open source, Nessie permet de gérer les tables Iceberg avec des branches, des tags et des commits atomiques multi-tables.

Cette approche apporte des avantages uniques pour la gouvernance. Les **branches** permettent d'isoler les environnements (développement, test, production) tout en partageant le même stockage physique. Les **tags** marquent des états stables (version auditée, point de conformité). Les **commits atomiques** garantissent la cohérence lors des modifications multi-tables.

```
# Configuration de Nessie avec contrôles d'accès basés sur les branches
from pynessie import init
from pynessie.auth import NessieAuthConfig
```

```
# Connexion au serveur Nessie avec authentification
nessie = init(
    "http://nessie.entreprise.ca:19120/api/v1",
    auth=NessieAuthConfig(
        type="bearer",
        token="${NESSIE_AUTH_TOKEN}"
    )
)

# Créer une branche pour l'équipe analytique
# Cette branche est en lecture seule pour la production
branch_analytique = nessie.create_branch(
    name="analytique-q1-2025",
    ref="main"
)

# Créer une branche pour les développements
branch_dev = nessie.create_branch(
    name="dev-feature-xyz",
    ref="main"
)

# Les permissions Nessie contrôlent :
# - Qui peut lire chaque branche
# - Qui peut committer sur chaque branche
# - Qui peut créer/supprimer des branches

# Configuration des règles d'accès (via fichier de configuration)
nessie_access_rules = """
authorization:
  rules:
    # Production : lecture seule sauf pour les data engineers
    - pattern: "main"
      actions:
        - allow: READ
          principals: ["group:analystes", "group:data_engineers"]
        - allow: COMMIT
          principals: ["group:data_engineers"]
        - allow: CREATE_REFERENCE
          principals: ["group:data_engineers"]

    # Branches analytiques : lecture seule
    - pattern: "analytique-*"
      actions:
        - allow: READ
          principals: ["group:analystes"]
        - allow: COMMIT
          principals: []  # Personne ne peut modifier

    # Branches de développement : accès complet pour les devs
    - pattern: "dev-*"
      actions:
        - allow: READ
          principals: ["group:developpeurs", "group:data_engineers"]
        - allow: COMMIT
          principals: ["group:developpeurs", "group:data_engineers"]
        - allow: DELETE_REFERENCE
          principals: ["group:data_engineers"]
"""
```

La gouvernance basée sur les branches offre un contrôle temporel unique. Un auditeur peut examiner l'état exact des données à un moment donné en créant un tag, sans risque que ces données soient modifiées. Les analystes peuvent travailler sur une branche stable pendant que les ingénieurs préparent des modifications sur une autre branche.

Gestion des Clés et Rotation

La gestion du cycle de vie des clés de chiffrement constitue un aspect critique souvent négligé. Une stratégie robuste doit couvrir la création, la distribution, la rotation et la révocation des clés.

La rotation automatique des clés limite l'impact d'une compromission potentielle. AWS KMS permet de configurer une rotation annuelle automatique des CMK. Cependant, cette rotation ne re-chiffre pas automatiquement les données existantes; les anciennes versions des clés sont conservées pour déchiffrer les données historiques.

```
# Stratégie de rotation des clés pour le Lakehouse
import boto3
from datetime import datetime, timedelta

class GestionnaireClésLakehouse:
    """
    Gère le cycle de vie des clés KMS pour le Lakehouse.

    Stratégie recommandée :
    - Rotation automatique annuelle via KMS
    - Re-chiffrement périodique des données sensibles
    - Séparation des clés par classification de données
    """

    def __init__(self, region: str = 'ca-central-1'):
        self.kms = boto3.client('kms', region_name=region)
        self.s3 = boto3.client('s3', region_name=region)

    def creer_cle_pour_classification(self,
                                      classification: str,
                                      description: str) -> str:
        """
        Crée une clé KMS dédiée à une classification de données.

        Classifications suggérées :
        - RESTREINT : Données hautement sensibles (NAS, financières)
        - CONFIDENTIEL : Données personnelles (PII)
        - INTERNE : Données métier non publiques
        - PUBLIC : Données sans restriction
        """
        response = self.kms.create_key(
            Description=f"Lakehouse - {description}",
            KeyUsage='ENCRYPT_DECRYPT',
            KeySpec='SYMMETRIC_DEFAULT',
            Origin='AWS_KMS',
            Tags=[
                {'TagKey': 'Environment', 'TagValue': 'production'},
                {'TagKey': 'Classification', 'TagValue': classification},
                {'TagKey': 'Application', 'TagValue': 'lakehouse'},
                {'TagKey': 'ManagedBy', 'TagValue': 'data-platform-team'}
            ]
        )
```

```
        key_id = response['KeyMetadata']['KeyId']

        # Activer la rotation automatique
        self.kms.enable_key_rotation(KeyId=key_id)

        # Créer un alias pour faciliter l'utilisation
        alias = f"alias/lakehouse-{classification.lower()}"
        self.kms.create_alias(
            AliasName=alias,
            TargetKeyId=key_id
        )

        return key_id

    def planifier_rechiffrement(self,
                                table_path: str,
                                spark_session) -> None:
        """
        Planifie le re-chiffrement d'une table Iceberg.

        Le re-chiffrement est nécessaire pour :
        - Révoquer l'accès aux anciennes versions de clés
        - Migrer vers des algorithmes plus forts
        - Conformité aux politiques de rotation
        """
        # Réécrire les fichiers de données avec la nouvelle version de clé
        spark_session.sql(f"""
            CALL system.rewrite_data_files(
                table => '{table_path}',
                options => map(
                    'rewrite-all', 'true',
                    'target-file-size-bytes', '268435456'
                )
            )
        """)

        # Expirer les anciens snapshots (chiffrés avec l'ancienne clé)
        spark_session.sql(f"""
            CALL system.expire_snapshots(
                table => '{table_path}',
                older_than => TIMESTAMP '{datetime.utcnow().isoformat()}',
                retain_last => 1
            )
        """)

        # Nettoyer les fichiers orphelins
        spark_session.sql(f"""
            CALL system.remove_orphan_files(
                table => '{table_path}'
            )
        """)

    def auditer_utilisation_cles(self) -> dict:
        """
        Audite l'utilisation des clés KMS pour identifier :
        - Clés non utilisées (candidates à la suppression)
        - Clés sur-utilisées (risque de performance)
        - Patterns d'accès anormaux
```

```
        """
        # Récupérer toutes les clés du Lakehouse
        paginator = self.kms.get_paginator('list_keys')

        rapport = {
            'cles_actives': [],
            'cles_desactivees': [],
            'cles_sans_rotation': [],
            'alertes': []
        }

        for page in paginator.paginate():
            for key in page['Keys']:
                key_id = key['KeyId']

                # Récupérer les métadonnées
                metadata = self.kms.describe_key(KeyId=key_id)['KeyMetadata']

                # Vérifier si c'est une clé Lakehouse
                tags = self.kms.list_resource_tags(KeyId=key_id).get('Tags', [])
                if not any(t['TagKey'] == 'Application' and
                           t['TagValue'] == 'lakehouse' for t in tags):
                    continue

                # Vérifier l'état
                if metadata['KeyState'] == 'Enabled':
                    rapport['cles_actives'].append(key_id)
                else:
                    rapport['cles_desactivees'].append(key_id)

                # Vérifier la rotation
                rotation_status = self.kms.get_key_rotation_status(KeyId=key_id)
                if not rotation_status['KeyRotationEnabled']:
                    rapport['cles_sans_rotation'].append(key_id)
                    rapport['alertes'].append(
                        f"ALERTE: Clé {key_id} n'a pas la rotation activée"
                    )

        return rapport
```

La séparation des clés par classification limite l'impact d'une compromission. Une clé compromise pour les données « INTERNE » ne permet pas d'accéder aux données « RESTREINT ». Cette approche complexifie la gestion mais renforce significativement la posture de sécurité.

IV.13.2 Contrôles d'Accès Granulaires

Les Niveaux de Contrôle d'Accès

La sécurité d'un Lakehouse mature repose sur plusieurs niveaux de contrôle d'accès complémentaires. Chaque niveau adresse des besoins différents et offre des compromis distincts entre granularité, performance et complexité de gestion.

Le Role-Based Access Control (RBAC) constitue le niveau fondamental. Il contrôle l'accès aux objets (catalogues, namespaces, tables, vues) en fonction des rôles attribués aux utilisateurs. RBAC répond à la question : « Cet utilisateur peut-il accéder à cette table ? ». Ce niveau est typiquement implémenté au niveau du catalogue et s'applique de manière cohérente à tous les moteurs de calcul.

Le Fine-Grained Access Control (FGAC) affine le contrôle au niveau des lignes et des colonnes. Il répond à des questions plus précises : « Cet utilisateur peut-il voir les lignes correspondant à sa région ? » ou « Cette colonne doit-elle être masquée pour cet utilisateur ? ». FGAC est généralement implémenté au niveau des moteurs de calcul car il nécessite l'évaluation des données elles-mêmes.

L'Attribute-Based Access Control (ABAC) généralise ces contrôles en définissant des politiques basées sur des attributs (tags, classifications) plutôt que sur des objets individuels. Une politique ABAC pourrait stipuler : « Toutes les colonnes taguées PII doivent être masquées pour les utilisateurs sans le rôle compliance ». Cette approche simplifie la gestion à grande échelle.

Row-Level Security (RLS)

La sécurité au niveau des lignes permet de filtrer automatiquement les données qu'un utilisateur peut voir en fonction de son identité ou de ses attributs. Cette capacité est essentielle pour les organisations où différentes équipes ou régions doivent accéder aux mêmes tables mais avec des périmètres de données différents.

Unity Catalog de Databricks offre une implémentation mature de RLS via les filtres de lignes. Ces filtres sont définis comme des fonctions SQL qui retournent TRUE pour les lignes accessibles à l'utilisateur courant.

```
-- Création d'une fonction de filtre pour la sécurité régionale
-- Les gestionnaires de compte ne voient que les clients de leur région
CREATE OR REPLACE FUNCTION production.securite.filtre_region_gestionnaire()
RETURNS BOOLEAN
RETURN
    is_account_group_member('administrateurs_donnees')  -- Accès complet
    OR current_user() IN (
        SELECT courriel_gestionnaire
        FROM production.reference.gestionnaires_regions gr
        WHERE gr.region_id = production.clients.transactions.region_id
    );

-- Application du filtre à la table des transactions
ALTER TABLE production.clients.transactions
SET ROW FILTER production.securite.filtre_region_gestionnaire ON ();

-- Vérification : Un gestionnaire de la région Québec
-- ne verra que les transactions de sa région
SELECT COUNT(*) FROM production.clients.transactions;
-- Retourne 45,230 (transactions Québec seulement)

-- Pour un administrateur de données
-- Retourne 1,234,567 (toutes les transactions)
```

AWS Lake Formation offre une approche similaire via les Data Filters. Ces filtres peuvent être définis de manière déclarative et appliqués lors de l'attribution des permissions.

```
# Configuration de Row-Level Security avec AWS Lake Formation
import boto3

lakeformation = boto3.client('lakeformation', region_name='ca-central-1')

# Créer un filtre de données pour les transactions
lakeformation.create_data_cells_filter(
    TableData={
        'DatabaseName': 'lakehouse',
        'Name': 'filtre_region_quebec',
        'TableCatalogId': '123456789012',
        'TableName': 'transactions',
        'RowFilter': {
            'FilterExpression': "region = 'QC'"
        }
    }
)

# Créer un filtre pour l'Ontario
lakeformation.create_data_cells_filter(
    TableData={
        'DatabaseName': 'lakehouse',
        'Name': 'filtre_region_ontario',
        'TableCatalogId': '123456789012',
        'TableName': 'transactions',
        'RowFilter': {
            'FilterExpression': "region = 'ON'"
        }
    }
)

# Attribuer les permissions avec le filtre
lakeformation.grant_permissions(
    Principal={
        'DataLakePrincipalIdentifier': 'arn:aws:iam::123456789012:role/AnalysteQuebec'
    },
    Resource={
        'DataCellsFilter': {
            'DatabaseName': 'lakehouse',
            'TableName': 'transactions',
            'TableCatalogId': '123456789012',
            'Name': 'filtre_region_quebec'
        }
    },
    Permissions=['SELECT']
)
```

Column-Level Security et Masquage Dynamique

Le masquage de colonnes permet de protéger les données sensibles sans créer de copies multiples des tables. Les utilisateurs autorisés voient les valeurs réelles, tandis que les autres voient des valeurs masquées ou nulles.

```
-- Fonction de masquage pour le numéro d'assurance sociale
CREATE OR REPLACE FUNCTION production.securite.masque_nas(nas STRING)
RETURNS STRING
```

```
RETURN
    CASE
        WHEN is_account_group_member('acces_nas_complet') THEN nas
        WHEN is_account_group_member('acces_nas_partiel') THEN
            CONCAT('***-***-', RIGHT(nas, 3))  -- Affiche seulement les 3 derniers
chiffres
        ELSE '***-***-***'  -- Masquage complet
    END;

-- Fonction de masquage pour les montants (arrondi pour certains utilisateurs)
CREATE OR REPLACE FUNCTION production.securite.masque_montant(montant DECIMAL(18,2))
RETURNS DECIMAL(18,2)
RETURN
    CASE
        WHEN is_account_group_member('finance_detail') THEN montant
        WHEN is_account_group_member('finance_apercu') THEN
            ROUND(montant, -3)  -- Arrondi au millier
        ELSE NULL
    END;

-- Application des masques à la table
ALTER TABLE production.clients.donnees_personnelles
ALTER COLUMN nas SET MASK production.securite.masque_nas;

ALTER TABLE production.finance.transactions
ALTER COLUMN montant SET MASK production.securite.masque_montant;

-- Résultat pour un analyste sans accès NAS complet
SELECT client_id, nom, nas, date_naissance
FROM production.clients.donnees_personnelles
LIMIT 3;

-- client_id | nom          | nas         | date_naissance
-- C001      | Tremblay     | ***-***-234 | 1985-03-15
-- C002      | Gagnon       | ***-***-891 | 1990-07-22
-- C003      | Roy          | ***-***-456 | 1978-11-08
```

Vues Dynamiques Sécurisées

Les vues dynamiques offrent une approche alternative au masquage au niveau des tables. Elles permettent de définir des transformations de sécurité complexes et de les appliquer de manière transparente aux consommateurs.

```
-- Vue sécurisée qui combine RLS et masquage de colonnes
CREATE OR REPLACE VIEW production.vues_securisees.v_clients_securise AS
SELECT
    client_id,
    nom,
    prenom,
    -- Masquage du courriel basé sur le groupe
    CASE
        WHEN is_account_group_member('marketing_complet') THEN courriel
        ELSE CONCAT(LEFT(courriel, 3), '***@', SPLIT_PART(courriel, '@', 2))
    END AS courriel,
    -- Masquage du téléphone
    CASE
```

```
        WHEN is_account_group_member('service_client') THEN telephone
        ELSE CONCAT('(***) ***-', RIGHT(telephone, 4))
    END AS telephone,
    -- NAS complètement masqué sauf pour compliance
    CASE
        WHEN is_account_group_member('compliance_pii') THEN nas
        ELSE NULL
    END AS nas,
    region,
    date_inscription,
    -- Segment client visible pour le marketing
    segment_client
FROM production.clients.donnees_personnelles
-- RLS : Filtre régional
WHERE
    is_account_group_member('acces_national')
    OR region = (
        SELECT region
        FROM production.reference.utilisateurs_regions
        WHERE courriel = current_user()
    );

-- Attribution des permissions sur la vue uniquement
GRANT SELECT ON production.vues_securisees.v_clients_securise
TO role_analyste_marketing;

-- Révocation de l'accès direct à la table source
REVOKE ALL ON production.clients.donnees_personnelles
FROM role_analyste_marketing;
```

Patterns de Masquage Avancés

Au-delà du masquage simple par remplacement de caractères, plusieurs patterns avancés permettent de préserver l'utilité analytique des données tout en protégeant la confidentialité.

Le masquage préservant le format maintient la structure des données (longueur, format) tout en remplaçant les valeurs. Cette approche est utile pour les tests et le développement où le réalisme des données est important.

```
-- Fonction de masquage préservant le format pour les numéros de carte
CREATE OR REPLACE FUNCTION production.securite.masque_carte_credit(
    numero_carte STRING
) RETURNS STRING
RETURN
    CASE
        WHEN is_account_group_member('paiements_complet') THEN numero_carte
        ELSE CONCAT(
            '****-****-****-',
            RIGHT(numero_carte, 4)  -- Préserve les 4 derniers chiffres
        )
    END;

-- Masquage préservant le format pour les codes postaux
CREATE OR REPLACE FUNCTION production.securite.masque_code_postal(
    code_postal STRING
) RETURNS STRING
```

```
RETURN
    CASE
        WHEN is_account_group_member('geo_complet') THEN code_postal
        -- Préserve la région (3 premiers caractères), masque le reste
        ELSE CONCAT(LEFT(code_postal, 3), ' ***')
    END;

-- Masquage avec bruit différentiel pour les montants
-- Permet l'analyse statistique tout en protégeant les valeurs individuelles
CREATE OR REPLACE FUNCTION production.securite.masque_montant_differentiel(
    montant DECIMAL(18,2),
    epsilon DOUBLE DEFAULT 0.1
) RETURNS DECIMAL(18,2)
RETURN
    CASE
        WHEN is_account_group_member('finance_detail') THEN montant
        ELSE ROUND(
            montant * (1 + (RAND() - 0.5) * epsilon),
            2
        )
    END;
```

La tokenisation remplace les valeurs sensibles par des jetons non réversibles, permettant les jointures et agrégations sans exposer les valeurs réelles.

```
-- Table de correspondance pour la tokenisation (accès très restreint)
CREATE TABLE production.securite.tokens_clients (
    token STRING,
    valeur_reelle STRING,  -- Chiffré avec clé séparée
    type_donnee STRING,    -- EMAIL, TELEPHONE, NAS
    date_creation TIMESTAMP,
    date_expiration TIMESTAMP
) USING iceberg;

-- Fonction de tokenisation
CREATE OR REPLACE FUNCTION production.securite.tokeniser(
    valeur STRING,
    type_donnee STRING
) RETURNS STRING
LANGUAGE PYTHON AS $$
    import hashlib
    import hmac

    # Clé secrète (en production, récupérée depuis un gestionnaire de secrets)
    secret_key = get_secret('tokenization_key')

    # Génération du token via HMAC
    token = hmac.new(
        secret_key.encode(),
        (type_donnee + ':' + valeur).encode(),
        hashlib.sha256
    ).hexdigest()[:16]

    return f"TKN_{type_donnee}_{token}"
$$;

-- Exemple d'utilisation : Vue tokenisée pour l'analytique
```

```
CREATE OR REPLACE VIEW production.analytique.v_transactions_tokenisees AS
SELECT
    transaction_id,
    production.securite.tokeniser(client_id, 'CLIENT') AS client_token,
    montant,
    date_transaction,
    categorie_produit,
    region
FROM production.gold.transactions;
```

La généralisation réduit la précision des données pour protéger les individus tout en préservant les tendances statistiques.

```
-- Généralisation de l'âge en tranches
CREATE OR REPLACE FUNCTION production.securite.generaliser_age(
    date_naissance DATE
) RETURNS STRING
RETURN
    CASE
        WHEN is_account_group_member('demographie_detail') THEN
            CAST(YEAR(current_date()) - YEAR(date_naissance) AS STRING)
        ELSE
            CASE
                WHEN YEAR(current_date()) - YEAR(date_naissance) < 18 THEN 'Moins de 18'
                WHEN YEAR(current_date()) - YEAR(date_naissance) < 25 THEN '18-24'
                WHEN YEAR(current_date()) - YEAR(date_naissance) < 35 THEN '25-34'
                WHEN YEAR(current_date()) - YEAR(date_naissance) < 45 THEN '35-44'
                WHEN YEAR(current_date()) - YEAR(date_naissance) < 55 THEN '45-54'
                WHEN YEAR(current_date()) - YEAR(date_naissance) < 65 THEN '55-64'
                ELSE '65 et plus'
            END
    END;

-- Généralisation géographique
CREATE OR REPLACE FUNCTION production.securite.generaliser_localisation(
    code_postal STRING
) RETURNS STRING
RETURN
    CASE
        WHEN is_account_group_member('geo_detail') THEN code_postal
        -- Retourne seulement la région de tri d'acheminement (RTA)
        ELSE LEFT(code_postal, 3)
    END;
```

> **Performance**
> Les contrôles d'accès granulaires (RLS, masquage) ajoutent une surcharge de traitement. Pour les tables très volumineuses, considérez les stratégies suivantes : - Utiliser des filtres basés sur des colonnes de partition pour bénéficier du pruning - Pré-calculer les tables de mapping des permissions pour éviter les sous-requêtes coûteuses - Évaluer l'utilisation de vues matérialisées pour les cas d'usage récurrents - Monitorer les temps de requête et ajuster les politiques si nécessaire

Tableaux de Bord de Gouvernance

La visibilité sur l'état de la gouvernance est essentielle pour maintenir une posture de sécurité adéquate. Un tableau de bord centralisé permet aux équipes de sécurité et de conformité de surveiller les métriques clés.

```
-- Métriques de gouvernance calculées quotidiennement
CREATE OR REPLACE VIEW production.gouvernance.v_metriques_quotidiennes AS
WITH metriques_acces AS (
    SELECT
        date_partition,
        COUNT(DISTINCT principal) AS utilisateurs_actifs,
        COUNT(*) AS total_requetes,
        SUM(CASE WHEN code_erreur = 'ACCESS_DENIED' THEN 1 ELSE 0 END) AS acces_refuses,
        COUNT(DISTINCT ressource_table) AS tables_accedees
    FROM lakehouse.audit.evenements_unifies
    WHERE date_partition >= current_date() - INTERVAL 30 DAYS
    GROUP BY date_partition
),

metriques_pii AS (
    SELECT
        date_partition,
        COUNT(*) AS acces_pii,
        COUNT(DISTINCT principal) AS utilisateurs_pii
    FROM lakehouse.audit.evenements_unifies
    WHERE ressource_table IN (
        SELECT chemin_technique
        FROM lakehouse.gouvernance.catalogue_metier
        WHERE contient_pii = true
    )
    AND date_partition >= current_date() - INTERVAL 30 DAYS
    GROUP BY date_partition
),

metriques_conformite AS (
    SELECT
        COUNT(*) AS demandes_effacement_en_cours,
        SUM(CASE WHEN statut = 'COMPLETE' THEN 1 ELSE 0 END) AS demandes_completees,
        AVG(delai_traitement_jours) AS delai_moyen_jours
    FROM lakehouse.conformite.demandes_effacement
    WHERE date_demande >= current_date() - INTERVAL 30 DAYS
)

SELECT
    ma.date_partition,
    ma.utilisateurs_actifs,
    ma.total_requetes,
    ma.acces_refuses,
    ROUND(100.0 * ma.acces_refuses / NULLIF(ma.total_requetes, 0), 2) AS taux_refus_pct,
    ma.tables_accedees,
    COALESCE(mp.acces_pii, 0) AS acces_pii,
    COALESCE(mp.utilisateurs_pii, 0) AS utilisateurs_pii,
    mc.demandes_effacement_en_cours,
    mc.demandes_completees,
    mc.delai_moyen_jours
FROM metriques_acces ma
LEFT JOIN metriques_pii mp ON ma.date_partition = mp.date_partition
CROSS JOIN metriques_conformite mc
```

```
ORDER BY ma.date_partition DESC;

-- Vue des anomalies de sécurité détectées
CREATE OR REPLACE VIEW production.gouvernance.v_anomalies_securite AS
SELECT
    timestamp,
    principal,
    type_anomalie,
    description,
    severite,
    statut_investigation
FROM (
    -- Accès en dehors des heures normales
    SELECT
        timestamp,
        principal,
        'ACCES_HORS_HEURES' AS type_anomalie,
        CONCAT('Accès à ', ressource_table, ' à ',
               DATE_FORMAT(timestamp, 'HH:mm')) AS description,
        'MOYEN' AS severite,
        'EN_ATTENTE' AS statut_investigation
    FROM lakehouse.audit.evenements_unifies
    WHERE HOUR(timestamp) < 6 OR HOUR(timestamp) > 22
    AND principal NOT IN (SELECT courriel FROM production.reference.comptes_service)

    UNION ALL

    -- Volume de données anormalement élevé
    SELECT
        MAX(timestamp) AS timestamp,
        principal,
        'VOLUME_ANORMAL' AS type_anomalie,
        CONCAT('Volume: ', SUM(lignes_affectees), ' lignes en 1 heure') AS description,
        CASE
            WHEN SUM(lignes_affectees) > 10000000 THEN 'CRITIQUE'
            WHEN SUM(lignes_affectees) > 1000000 THEN 'ELEVE'
            ELSE 'MOYEN'
        END AS severite,
        'EN_ATTENTE' AS statut_investigation
    FROM lakehouse.audit.evenements_unifies
    WHERE timestamp >= current_timestamp() - INTERVAL 1 HOUR
    GROUP BY principal
    HAVING SUM(lignes_affectees) > 100000

    UNION ALL

    -- Tentatives d'accès répétées refusées
    SELECT
        MAX(timestamp) AS timestamp,
        principal,
        'TENTATIVES_REFUSEES' AS type_anomalie,
        CONCAT(COUNT(*), ' tentatives refusées en 10 minutes') AS description,
        CASE
            WHEN COUNT(*) > 50 THEN 'CRITIQUE'
            WHEN COUNT(*) > 20 THEN 'ELEVE'
            ELSE 'MOYEN'
        END AS severite,
        'EN_ATTENTE' AS statut_investigation
    FROM lakehouse.audit.evenements_unifies
```

```
    WHERE code_erreur = 'ACCESS_DENIED'
    AND timestamp >= current_timestamp() - INTERVAL 10 MINUTES
    GROUP BY principal
    HAVING COUNT(*) > 10
) anomalies
WHERE timestamp >= current_date() - INTERVAL 7 DAYS
ORDER BY
    CASE severite
        WHEN 'CRITIQUE' THEN 1
        WHEN 'ELEVE' THEN 2
        ELSE 3
    END,
    timestamp DESC;

---

## IV.13.3 Gouvernance des Données

### Le Lignage des Données avec OpenLineage

Le lignage des données répond à des questions fondamentales pour la gouvernance : D'où viennent ces données ? Comment ont-elles été transformées ? Qui y a accédé ? Ces informations sont essentielles pour l'audit, l'analyse d'impact et le débogage des problèmes de qualité.

**OpenLineage** émerge comme le standard ouvert pour la collecte du lignage. Cette spécification, hébergée par la Linux Foundation AI & Data, définit un modèle de données et un protocole pour capturer les événements de lignage à travers différents outils et plateformes. Apache Spark, Apache Flink, Apache Airflow, dbt et de nombreux autres outils supportent désormais l'émission d'événements OpenLineage.

Le modèle OpenLineage capture trois entités principales. Les **Jobs** représentent les processus de transformation (un job Spark, un DAG Airflow, un modèle dbt). Les **Runs** représentent les exécutions individuelles de ces jobs. Les **Datasets** représentent les entrées et sorties (tables Iceberg, fichiers, topics Kafka).

```python
# Configuration OpenLineage pour Spark avec tables Iceberg
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("Pipeline-ETL-Clients") \
    .config("spark.jars.packages",
            "io.openlineage:openlineage-spark_2.12:1.23.0") \
    .config("spark.extraListeners",
            "io.openlineage.spark.agent.OpenLineageSparkListener") \
    .config("spark.openlineage.transport.type", "http") \
    .config("spark.openlineage.transport.url",
            "http://marquez.entreprise.ca:5000") \
    .config("spark.openlineage.namespace", "lakehouse-production") \
    .config("spark.openlineage.parentJobNamespace", "orchestration") \
    .config("spark.openlineage.parentJobName", "pipeline-clients-quotidien") \
    .getOrCreate()

# Exécution d'un pipeline ETL
# OpenLineage capture automatiquement les entrées, sorties et transformations

# Lecture depuis les sources
df_transactions = spark.read \
```
```

```
    .format("iceberg") \
    .load("lakehouse.bronze.transactions_brutes")

df_clients = spark.read \
    .format("iceberg") \
    .load("lakehouse.bronze.clients_bruts")

# Transformations
df_enrichi = df_transactions \
    .join(df_clients, "client_id") \
    .withColumn("montant_cad",
                F.when(F.col("devise") == "USD",
                       F.col("montant") * 1.36)
                 .otherwise(F.col("montant"))) \
    .withColumn("date_traitement", F.current_timestamp())

# Écriture vers la couche Silver
df_enrichi.writeTo("lakehouse.silver.transactions_enrichies") \
    .using("iceberg") \
    .partitionedBy(F.years("date_transaction")) \
    .createOrReplace()

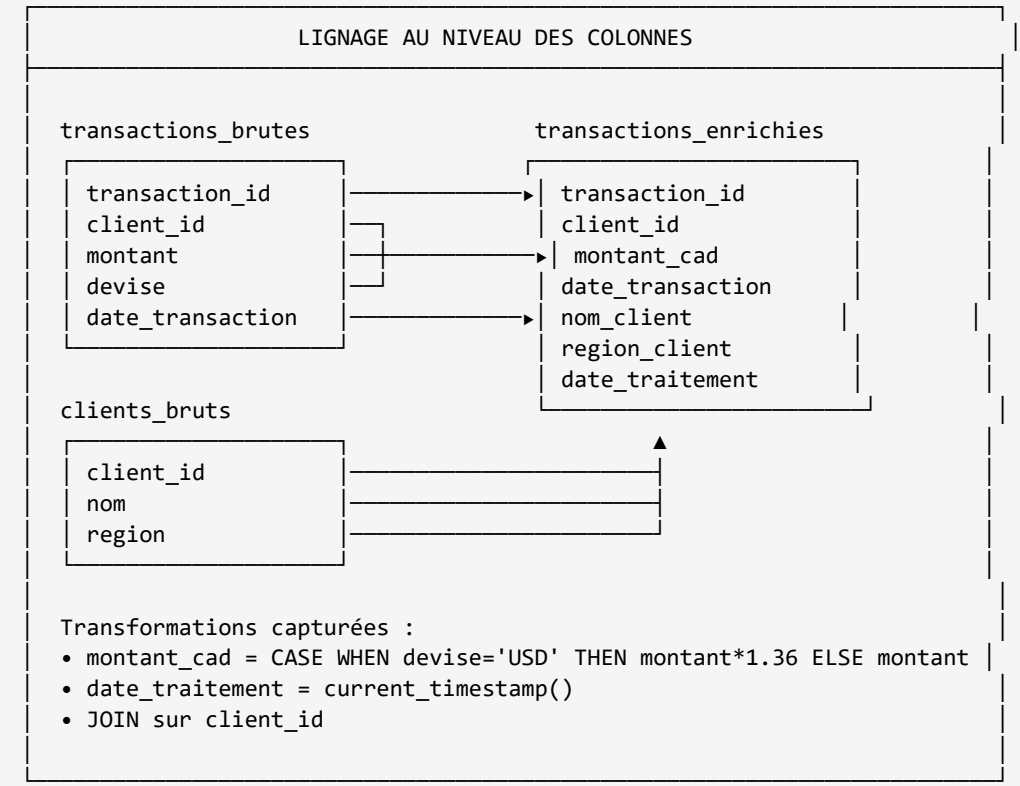
# OpenLineage émet automatiquement un événement incluant :
# - Input datasets : transactions_brutes, clients_bruts
# - Output dataset : transactions_enrichies
# - Schema de chaque dataset
# - Statistiques d'exécution (lignes lues/écrites, durée)
# - Lignage au niveau des colonnes (column-level lineage)
```

Marquez constitue l'implémentation de référence pour la collecte et la visualisation des événements OpenLineage. Il fournit une API REST pour l'ingestion, une base de données pour le stockage et une interface web pour l'exploration du graphe de lignage.

```yaml
# Configuration Docker Compose pour Marquez avec stockage PostgreSQL
version: '3.8'

services:
  marquez:
    image: marquezproject/marquez:latest
    ports:
      - "5000:5000"  # API
      - "5001:5001"  # Admin
    environment:
      MARQUEZ_CONFIG: /opt/marquez/config/marquez.yml
    volumes:
      - ./marquez-config.yml:/opt/marquez/config/marquez.yml
    depends_on:
      - postgres-marquez

  marquez-web:
    image: marquezproject/marquez-web:latest
    ports:
      - "3000:3000"
    environment:
      MARQUEZ_HOST: marquez
      MARQUEZ_PORT: 5000
```

```
  postgres-marquez:
    image: postgres:14
    environment:
      POSTGRES_USER: marquez
      POSTGRES_PASSWORD: ${MARQUEZ_DB_PASSWORD}
      POSTGRES_DB: marquez
    volumes:
      - marquez-data:/var/lib/postgresql/data

volumes:
  marquez-data:
```

Le lignage au niveau des colonnes (Column-Level Lineage) représente une avancée significative d'OpenLineage. Il permet de tracer exactement quelles colonnes source ont contribué à chaque colonne de sortie, incluant les transformations appliquées.



Qualité des Données

La gouvernance du Lakehouse inclut nécessairement la qualité des données. Les tables Iceberg peuvent contenir des milliards d'enregistrements, et détecter les anomalies de qualité devient un défi à cette échelle.

Plusieurs frameworks permettent d'implémenter des contrôles de qualité automatisés. **Great Expectations** offre une approche déclarative pour définir et valider des « attentes » sur les données. **Deequ** (Amazon) s'intègre nativement à Spark pour des validations performantes à grande échelle. **Soda Core** propose une syntaxe simple et des intégrations avec les orchestrateurs.

```python
# Validation de qualité avec Great Expectations sur une table Iceberg
import great_expectations as gx
from great_expectations.core.batch import RuntimeBatchRequest

# Configuration du contexte
context = gx.get_context()

# Définition de la suite de validations
context.add_expectation_suite("transactions_silver_suite")

# Création des attentes
validator = context.get_validator(
    batch_request=RuntimeBatchRequest(
        datasource_name="spark_iceberg",
        data_connector_name="default_runtime_data_connector",
        data_asset_name="transactions_enrichies",
        runtime_parameters={"path": "lakehouse.silver.transactions_enrichies"},
        batch_identifiers={"batch_id": "daily_2025_01_16"},
    ),
    expectation_suite_name="transactions_silver_suite",
)

# Attentes de complétude
validator.expect_column_values_to_not_be_null("transaction_id")
validator.expect_column_values_to_not_be_null("client_id")
validator.expect_column_values_to_not_be_null("montant_cad")

# Attentes de format
validator.expect_column_values_to_match_regex(
    "client_id",
    regex=r"^C[0-9]{6}$",
    meta={"description": "Format client_id : C suivi de 6 chiffres"}
)

# Attentes de plage
validator.expect_column_values_to_be_between(
    "montant_cad",
    min_value=0.01,
    max_value=1000000.00,
    meta={"description": "Montant entre 0.01 et 1M CAD"}
)

# Attentes de référentiel
validator.expect_column_values_to_be_in_set(
    "region_client",
    ["QC", "ON", "BC", "AB", "MB", "SK", "NS", "NB", "PE", "NL", "NT", "YT", "NU"]
)

# Attentes statistiques
validator.expect_column_mean_to_be_between(
    "montant_cad",
    min_value=50.0,
    max_value=500.0,
    meta={"description": "Montant moyen attendu entre 50 et 500 CAD"}
)

# Unicité
validator.expect_column_values_to_be_unique("transaction_id")
```

```
# Fraîcheur des données
validator.expect_column_max_to_be_between(
    "date_transaction",
    min_value=(datetime.now() - timedelta(days=1)).isoformat(),
    max_value=datetime.now().isoformat(),
    meta={"description": "Données datant de moins de 24h"}
)

# Exécution des validations
results = validator.validate()

# Intégration avec le pipeline : échec si validations critiques échouent
if not results.success:
    critical_failures = [
        r for r in results.results
        if not r.success and r.expectation_config.meta.get("severity") == "critical"
    ]
    if critical_failures:
        raise DataQualityException(
            f"Échec des validations critiques : {critical_failures}"
        )
```

Catalogage et Découverte des Données

Un catalogue de données métier complète le catalogue technique (Polaris, Glue) en ajoutant une couche de contexte métier. Il permet aux utilisateurs de découvrir les données disponibles, comprendre leur signification et identifier les propriétaires.

```
-- Structure de métadonnées métier dans le Lakehouse
-- Table de documentation des assets de données

CREATE TABLE lakehouse.gouvernance.catalogue_metier (
    asset_id STRING,
    asset_type STRING,  -- TABLE, VIEW, MODEL, DASHBOARD
    chemin_technique STRING,  -- lakehouse.silver.transactions
    nom_metier STRING,
    description STRING,
    domaine_metier STRING,  -- Finance, Marketing, Operations
    proprietaire_courriel STRING,
    intendant_donnees_courriel STRING,
    classification_donnees STRING,  -- PUBLIC, INTERNE, CONFIDENTIEL, RESTREINT
    contient_pii BOOLEAN,
    contient_donnees_financieres BOOLEAN,
    retention_jours INT,
    reglementation_applicable ARRAY<STRING>,  -- ['LOI25', 'LPRPDE', 'SOX']
    termes_glossaire ARRAY<STRING>,
    tags ARRAY<STRING>,
    date_creation TIMESTAMP,
    date_derniere_modification TIMESTAMP,
    frequence_mise_a_jour STRING,  -- TEMPS_REEL, QUOTIDIEN, HEBDOMADAIRE
    documentation_url STRING
) USING iceberg
PARTITIONED BY (domaine_metier);

-- Exemple d'entrée de catalogue
INSERT INTO lakehouse.gouvernance.catalogue_metier VALUES (
```

```
    'asset_001',
    'TABLE',
    'lakehouse.silver.transactions_enrichies',
    'Transactions Clients Enrichies',
    'Table consolidée des transactions clients avec enrichissement démographique et
conversion de devise. Source de vérité pour les rapports financiers et analyses
marketing.',
    'Finance',
    'marie.tremblay@entreprise.ca',
    'jean.gagnon@entreprise.ca',
    'CONFIDENTIEL',
    true,
    true,
    2555,  -- 7 ans pour conformité SOX
    array('LOI25', 'LPRPDE', 'SOX'),
    array('transaction', 'client', 'montant', 'chiffre_affaires'),
    array('finance', 'analytique', 'certifie'),
    current_timestamp(),
    current_timestamp(),
    'QUOTIDIEN',
    'https://wiki.entreprise.ca/donnees/transactions-enrichies'
);
```

IV.13.4 Conformité Réglementaire au Canada

La Loi 25 du Québec

La Loi 25 (anciennement Projet de loi 64), pleinement en vigueur depuis septembre 2024, constitue le cadre de protection des renseignements personnels le plus strict au Canada. Elle s'aligne sur le RGPD européen et impose des obligations significatives aux organisations qui traitent des données de résidents québécois.

Les principales exigences de la Loi 25 avec impact sur l'architecture Lakehouse incluent :

Le consentement explicite pour la collecte et l'utilisation des renseignements personnels. Le Lakehouse doit pouvoir démontrer que le consentement a été obtenu et enregistrer les préférences de chaque individu.

Le droit à l'effacement (droit à l'oubli) permet aux individus de demander la suppression de leurs données. Cette exigence entre en tension directe avec l'immutabilité naturelle d'un Lakehouse, où le time travel conserve l'historique des données.

Le droit à la portabilité oblige les organisations à fournir les données personnelles dans un format structuré et couramment utilisé. Le Lakehouse doit pouvoir extraire et formater les données d'un individu spécifique.

L'évaluation des facteurs relatifs à la vie privée (EFVP) est obligatoire pour tout projet impliquant des renseignements personnels. L'architecture du Lakehouse elle-même doit faire l'objet d'une EFVP.

```
# Implémentation du droit à l'effacement conforme à la Loi 25
from datetime import datetime
from dataclasses import dataclass
from typing import List, Dict, Optional
import logging
```

```
@dataclass
class DemandeEffacement:
    """Représente une demande d'effacement Loi 25."""
    demande_id: str
    identifiant_personne: str  # Courriel, NAS, ou autre identifiant
    type_identifiant: str
    date_demande: datetime
    tables_a_traiter: List[str]
    demandeur_courriel: str
    motif: str
    statut: str = "EN_ATTENTE"

class GestionnaireDroitEffacementLoi25:
    """
    Gestionnaire des demandes d'effacement conformes à la Loi 25.

    Responsabilités :
    - Traiter les demandes d'effacement dans le délai légal (30 jours)
    - Supprimer les données des tables Iceberg
    - Expirer les snapshots contenant les données supprimées
    - Produire les tombstones Kafka pour la propagation
    - Générer le certificat de suppression pour audit
    """

    def __init__(self, spark_session, kafka_producer,
                 iceberg_catalog, journal_audit):
        self.spark = spark_session
        self.producer = kafka_producer
        self.catalog = iceberg_catalog
        self.audit = journal_audit
        self.logger = logging.getLogger(__name__)

    def traiter_demande(self, demande: DemandeEffacement) -> Dict:
        """
        Traite une demande d'effacement de bout en bout.

        Étapes :
        1. Validation et journalisation de la demande
        2. Identification des données à supprimer
        3. Suppression des tables Iceberg (DELETE)
        4. Expiration des snapshots historiques
        5. Production des tombstones Kafka
        6. Génération du certificat de conformité
        """
        self.logger.info(f"Traitement demande effacement: {demande.demande_id}")

        # 1. Journaliser la demande pour audit
        self.audit.journaliser_demande_effacement(
            demande_id=demande.demande_id,
            identifiant=demande.identifiant_personne,
            type_identifiant=demande.type_identifiant,
            date_demande=demande.date_demande,
            demandeur=demande.demandeur_courriel
        )

        resultats = {}

        # 2-3. Pour chaque table, supprimer les données
```

```
        for table_name in demande.tables_a_traiter:
            try:
                # Identifier la colonne correspondant à l'identifiant
                colonne_id = self._obtenir_colonne_identifiant(
                    table_name,
                    demande.type_identifiant
                )

                # Compter les lignes avant suppression (pour audit)
                count_avant = self._compter_lignes(
                    table_name, colonne_id, demande.identifiant_personne
                )

                # Exécuter la suppression via Spark SQL
                self.spark.sql(f"""
                    DELETE FROM {table_name}
                    WHERE {colonne_id} = '{demande.identifiant_personne}'
                """)

                # Vérifier la suppression
                count_apres = self._compter_lignes(
                    table_name, colonne_id, demande.identifiant_personne
                )

                resultats[table_name] = {
                    'lignes_supprimees': count_avant - count_apres,
                    'timestamp': datetime.utcnow().isoformat(),
                    'statut': 'SUCCES' if count_apres == 0 else 'PARTIEL'
                }

                # Journaliser pour audit
                self.audit.journaliser_suppression_table(
                    demande_id=demande.demande_id,
                    table=table_name,
                    lignes_supprimees=count_avant - count_apres
                )

            except Exception as e:
                self.logger.error(f"Erreur suppression {table_name}: {e}")
                resultats[table_name] = {
                    'erreur': str(e),
                    'statut': 'ECHEC'
                }

        # 4. Expirer les snapshots historiques contenant les données
        self._expirer_snapshots_historiques(demande.tables_a_traiter)

        # 5. Produire les tombstones Kafka pour propagation
        self._produire_tombstones(
            identifiant=demande.identifiant_personne,
            type_identifiant=demande.type_identifiant
        )

        # 6. Générer le certificat de suppression
        certificat = self._generer_certificat(demande, resultats)

        return certificat

    def _expirer_snapshots_historiques(self, tables: List[str]) -> None:
```

```
        """
        Expire les snapshots Iceberg pour garantir que les données
        supprimées ne sont plus accessibles via time travel.

        Note : Cette opération doit être planifiée avec soin car
        elle est irréversible et affecte la capacité de rollback.
        """
        for table_name in tables:
            # Expirer tous les snapshots antérieurs à maintenant
            self.spark.sql(f"""
                CALL system.expire_snapshots(
                    table => '{table_name}',
                    older_than => TIMESTAMP '{datetime.utcnow().isoformat()}',
                    retain_last => 1
                )
            """)

            # Nettoyer les fichiers orphelins
            self.spark.sql(f"""
                CALL system.remove_orphan_files(
                    table => '{table_name}'
                )
            """)

    def _produire_tombstones(self, identifiant: str,
                             type_identifiant: str) -> None:
        """
        Produit des tombstones Kafka pour les topics contenant
        potentiellement les données de la personne concernée.

        Un tombstone (message avec valeur null) permet la compaction
        du log Kafka pour supprimer les données historiques.
        """
        topics_affectes = self._identifier_topics_affectes(type_identifiant)

        for topic in topics_affectes:
            self.producer.send(
                topic=topic,
                key=identifiant.encode('utf-8'),
                value=None  # Tombstone
            )
            self.producer.flush()

    def _generer_certificat(self, demande: DemandeEffacement,
                            resultats: Dict) -> Dict:
        """
        Génère un certificat de suppression attestant de l'exécution
        de la demande d'effacement, requis pour audit Loi 25.
        """
        certificat = {
            'certificat_id': f"CERT-{demande.demande_id}",
            'demande_id': demande.demande_id,
            'date_traitement': datetime.utcnow().isoformat(),
            'identifiant_personne_hash': self._hasher_identifiant(
                demande.identifiant_personne
            ),
            'resultats_par_table': resultats,
            'snapshots_expires': True,
            'tombstones_produits': True,
```

```
            'conformite_loi25': True,
            'delai_traitement_jours': (
                datetime.utcnow() - demande.date_demande
            ).days
        }

        # Vérification du délai légal (30 jours)
        if certificat['delai_traitement_jours'] > 30:
            certificat['alerte_delai'] = "DEPASSEMENT_DELAI_LEGAL"
            self.logger.warning(
                f"Demande {demande.demande_id} traitée hors délai légal"
            )

        # Stocker le certificat pour audit
        self.audit.stocker_certificat(certificat)

        return certificat
```

La LPRPDE Fédérale

La Loi sur la protection des renseignements personnels et les documents électroniques (LPRPDE) s'applique aux organisations du secteur privé dans tout le Canada pour les activités commerciales. Bien que moins stricte que la Loi 25, elle établit des principes fondamentaux que le Lakehouse doit respecter.

Les dix principes de la LPRPDE se traduisent en exigences architecturales :

Principe LPRPDE	Implication pour le Lakehouse
Responsabilité	Désignation d'un responsable, documentation des politiques
Détermination des fins	Métadonnées documentant la finalité de chaque dataset
Consentement	Enregistrement des consentements, respect des préférences
Limitation de la collecte	Contrôles limitant les données collectées au nécessaire
Limitation de l'utilisation	Contrôles d'accès basés sur la finalité déclarée
Exactitude	Processus de validation et correction des données
Mesures de sécurité	Chiffrement, contrôles d'accès, surveillance
Transparence	Catalogue accessible documentant les données détenues
Accès individuel	Mécanismes d'extraction des données personnelles
Possibilité de porter plainte	Processus de gestion des demandes et plaintes

Conformité Sectorielle (BSIF, SOX)

Les organisations dans des secteurs réglementés font face à des exigences supplémentaires. Le Bureau du surintendant des institutions financières (BSIF) impose aux banques et assureurs canadiens des directives strictes en matière de gestion des risques technologiques et de résilience opérationnelle.

La directive B-13 (Gestion du risque lié aux technologies et du cyberrisque) exige notamment l'inventaire complet des actifs de données, la classification selon leur criticité, et des contrôles d'accès proportionnés aux risques.

La ligne directrice E-21 (Gestion du risque lié aux tiers) s'applique lorsque le Lakehouse utilise des services infonuagiques de tiers (AWS, Azure, GCP, Confluent). Elle exige une diligence raisonnable, des clauses contractuelles appropriées et une surveillance continue.

```
-- Structure de classification des données pour conformité BSIF
CREATE TABLE lakehouse.gouvernance.classification_bsif (
    table_id STRING,
    chemin_table STRING,
    classification_criticite STRING,  -- CRITIQUE, IMPORTANT, STANDARD
    classification_sensibilite STRING,  -- TRES_ELEVE, ELEVE, MOYEN, FAIBLE
    proprietaire_metier STRING,
    gardien_technique STRING,

    -- Exigences de protection basées sur classification
    chiffrement_requis BOOLEAN,
    masquage_requis BOOLEAN,
    retention_minimale_jours INT,
    retention_maximale_jours INT,

    -- Contrôles d'accès
    acces_restreint_equipes ARRAY<STRING>,
    approbation_requise BOOLEAN,
    niveau_approbation STRING,  -- GESTIONNAIRE, DIRECTEUR, VP

    -- Résilience
    rpo_heures INT,  -- Recovery Point Objective
    rto_heures INT,  -- Recovery Time Objective
    sauvegarde_requise BOOLEAN,
    replication_geo_requise BOOLEAN,

    -- Audit
    audit_acces_requis BOOLEAN,
    retention_audit_jours INT,

    -- Tiers
    partage_tiers_autorise BOOLEAN,
    tiers_autorises ARRAY<STRING>,

    date_derniere_revue DATE,
    prochaine_revue DATE
) USING iceberg;

-- Exemple de classification pour une table critique
INSERT INTO lakehouse.gouvernance.classification_bsif VALUES (
    'tbl_001',
    'lakehouse.gold.positions_portefeuille',
    'CRITIQUE',
    'TRES_ELEVE',
    'vp.gestion.actifs@banque.ca',
    'architecte.donnees@banque.ca',
    true,   -- chiffrement
    true,   -- masquage
    2555,   -- retention min 7 ans
    3650,   -- retention max 10 ans
```

```
    array('equipe_gestion_actifs', 'equipe_risques', 'equipe_conformite'),
    true,   -- approbation requise
    'VP',
    4,      -- RPO 4 heures
    24,     -- RTO 24 heures
    true,   -- sauvegarde
    true,   -- réplication géo
    true,   -- audit accès
    2555,   -- retention audit 7 ans
    false,  -- pas de partage tiers
    array(),
    current_date(),
    date_add(current_date(), 365)
);
```

IV.13.5 Audit et Surveillance

Journalisation des Accès

Une stratégie d'audit complète capture les accès à plusieurs niveaux : le stockage objet, le catalogue et les moteurs de calcul. La corrélation de ces journaux permet de reconstituer l'activité complète d'un utilisateur.

```
# Pipeline d'agrégation des journaux d'audit multi-sources
from pyspark.sql import SparkSession
from pyspark.sql import functions as F

spark = SparkSession.builder \
    .appName("Audit-Aggregation") \
    .getOrCreate()

# Source 1 : AWS CloudTrail (accès S3)
df_cloudtrail = spark.read \
    .format("json") \
    .load("s3://audit-logs/cloudtrail/") \
    .filter(F.col("eventSource") == "s3.amazonaws.com") \
    .select(
        F.col("eventTime").alias("timestamp"),
        F.lit("S3").alias("source"),
        F.col("userIdentity.arn").alias("principal"),
        F.col("eventName").alias("action"),
        F.col("requestParameters.bucketName").alias("ressource_bucket"),
        F.col("requestParameters.key").alias("ressource_cle"),
        F.col("sourceIPAddress").alias("ip_source"),
        F.col("errorCode").alias("code_erreur")
    )

# Source 2 : Polaris Audit Logs (accès catalogue)
df_polaris = spark.read \
    .format("json") \
    .load("s3://audit-logs/polaris/") \
    .select(
```

```
        F.col("timestamp"),
        F.lit("POLARIS").alias("source"),
        F.col("principal"),
        F.col("action"),
        F.col("catalog").alias("ressource_catalogue"),
        F.col("namespace").alias("ressource_namespace"),
        F.col("table").alias("ressource_table"),
        F.col("sourceIp").alias("ip_source"),
        F.col("errorCode").alias("code_erreur")
    )

# Source 3 : Spark Query Logs (requêtes exécutées)
df_spark_audit = spark.read \
    .format("json") \
    .load("s3://audit-logs/spark/") \
    .select(
        F.col("timestamp"),
        F.lit("SPARK").alias("source"),
        F.col("user").alias("principal"),
        F.col("operation").alias("action"),
        F.col("database").alias("ressource_database"),
        F.col("table").alias("ressource_table"),
        F.col("query").alias("requete_sql"),
        F.col("duration_ms").alias("duree_ms"),
        F.col("rows_affected").alias("lignes_affectees")
    )

# Agrégation et enrichissement
df_audit_unifie = df_cloudtrail \
    .unionByName(df_polaris, allowMissingColumns=True) \
    .unionByName(df_spark_audit, allowMissingColumns=True) \
    .withColumn("date_partition", F.to_date("timestamp")) \
    .withColumn("heure", F.hour("timestamp"))

# Écriture vers la table d'audit unifiée
df_audit_unifie.writeTo("lakehouse.audit.evenements_unifies") \
    .using("iceberg") \
    .partitionedBy("date_partition") \
    .append()

# Création de vues analytiques pour les investigations
spark.sql("""
    CREATE OR REPLACE VIEW lakehouse.audit.v_acces_pii AS
    SELECT
        timestamp,
        source,
        principal,
        action,
        COALESCE(ressource_table, ressource_cle) as ressource,
        ip_source,
        requete_sql
    FROM lakehouse.audit.evenements_unifies
    WHERE
        ressource_table IN (
            SELECT chemin_technique
            FROM lakehouse.gouvernance.catalogue_metier
            WHERE contient_pii = true
        )
        OR ressource_cle LIKE '%donnees_personnelles%'
```

```
    ORDER BY timestamp DESC
""")
```

Alertes de Sécurité

La détection proactive des comportements anormaux constitue une couche de défense essentielle. Les patterns suivants méritent une surveillance particulière :

```
# Configuration des alertes de sécurité pour le Lakehouse
# alertmanager-rules.yml

groups:
  - name: lakehouse_security_alerts
    rules:
      # Tentatives d'accès non autorisées
      - alert: AccesNonAutoriseDetecte
        expr: |
          sum(rate(polaris_access_denied_total[5m])) by (principal) > 10
        for: 5m
        labels:
          severity: warning
          equipe: securite
        annotations:
          summary: "Tentatives d'accès refusées répétées"
          description: "Le principal {{ $labels.principal }} a eu {{ $value }} tentatives
refusées en 5 minutes"

      # Accès inhabituel aux données PII
      - alert: AccesPIIAnormal
        expr: |
          sum(rate(spark_query_pii_tables_total[1h])) by (user)
          > 3 * avg_over_time(spark_query_pii_tables_total[7d])
        for: 15m
        labels:
          severity: high
          equipe: securite
        annotations:
          summary: "Volume d'accès PII anormalement élevé"
          description: "L'utilisateur {{ $labels.user }} accède aux tables PII à un rythme
3x supérieur à la normale"

      # Export massif de données
      - alert: ExportMassifDetecte
        expr: |
          sum(rate(iceberg_rows_read_total[10m])) by (user, table) > 10000000
        for: 5m
        labels:
          severity: critical
          equipe: securite
        annotations:
          summary: "Export massif de données détecté"
          description: "{{ $labels.user }} lit >10M lignes de {{ $labels.table }} en 10
minutes"

      # Accès depuis IP inhabituelle
      - alert: AccesIPInhabituelle
```

```
        expr: |
          count(polaris_access_total{ip_source!~"10\\..*|172\\.16\\..*|192\\.168\\..*"})
by (principal, ip_source) > 0
        for: 1m
        labels:
          severity: high
          equipe: securite
        annotations:
          summary: "Accès depuis IP externe"
          description: "{{ $labels.principal }} accède depuis l'IP
{{ $labels.ip_source }}"

      # Modification de permissions
      - alert: ModificationPermissions
        expr: |
          increase(polaris_grant_revoke_total[5m]) > 0
        for: 0m
        labels:
          severity: info
          equipe: gouvernance
        annotations:
          summary: "Modification des permissions détectée"
          description: "Des permissions ont été modifiées dans le catalogue"
```

Réponse aux Incidents de Sécurité

Un plan de réponse aux incidents documenté et testé est essentiel pour minimiser l'impact d'une violation de données. Le Lakehouse, grâce à ses capacités de time travel et son audit détaillé, offre des outils puissants pour l'investigation et la remédiation.

```python
# Framework de réponse aux incidents pour le Lakehouse
from dataclasses import dataclass
from datetime import datetime
from enum import Enum
from typing import List, Optional
import json

class SeveriteIncident(Enum):
    CRITIQUE = "CRITIQUE"  # Données sensibles exposées, action immédiate
    ELEVE = "ELEVE"        # Accès non autorisé détecté
    MOYEN = "MOYEN"        # Anomalie nécessitant investigation
    FAIBLE = "FAIBLE"      # Événement suspect à surveiller

@dataclass
class IncidentSecurite:
    """Représente un incident de sécurité dans le Lakehouse."""
    incident_id: str
    timestamp_detection: datetime
    severite: SeveriteIncident
    type_incident: str  # ACCES_NON_AUTORISE, EXFILTRATION, MODIFICATION
    description: str
    tables_affectees: List[str]
    utilisateur_suspect: Optional[str]
    ip_source: Optional[str]
    statut: str = "OUVERT"
```

```
class GestionnaireIncidents:
    """
    Gère le cycle de vie des incidents de sécurité du Lakehouse.

    Phases de réponse :
    1. Détection et qualification
    2. Confinement immédiat
    3. Investigation approfondie
    4. Remédiation
    5. Récupération
    6. Leçons apprises
    """

    def __init__(self, spark_session, polaris_client,
                 notification_service):
        self.spark = spark_session
        self.polaris = polaris_client
        self.notifier = notification_service

    def phase_1_detection(self, incident: IncidentSecurite) -> dict:
        """
        Phase 1 : Détection et qualification initiale.

        Actions :
        - Confirmer l'incident
        - Évaluer la sévérité
        - Notifier les parties prenantes appropriées
        """
        # Journaliser l'incident
        self._journaliser_incident(incident)

        # Notification selon la sévérité
        if incident.severite == SeveriteIncident.CRITIQUE:
            self.notifier.alerte_urgente(
                destinataires=["securite@entreprise.ca",
                               "ciso@entreprise.ca",
                               "juridique@entreprise.ca"],
                sujet=f"[CRITIQUE] Incident sécurité Lakehouse: {incident.incident_id}",
                corps=self._formater_alerte(incident)
            )
        elif incident.severite == SeveriteIncident.ELEVE:
            self.notifier.alerte_urgente(
                destinataires=["securite@entreprise.ca"],
                sujet=f"[ÉLEVÉ] Incident sécurité Lakehouse: {incident.incident_id}",
                corps=self._formater_alerte(incident)
            )

        return {"phase": "DETECTION", "statut": "COMPLETE"}

    def phase_2_confinement(self, incident: IncidentSecurite) -> dict:
        """
        Phase 2 : Confinement pour limiter l'impact.

        Actions :
        - Révoquer les accès suspects
        - Isoler les ressources compromises
        - Préserver les preuves
        """
        actions_prises = []
```

```
        # Révoquer immédiatement les accès de l'utilisateur suspect
        if incident.utilisateur_suspect:
            self.polaris.revoquer_tous_acces(
                principal=incident.utilisateur_suspect
            )
            actions_prises.append(
                f"Accès révoqués pour {incident.utilisateur_suspect}"
            )

        # Bloquer l'IP source si identifiée
        if incident.ip_source:
            self._ajouter_ip_liste_noire(incident.ip_source)
            actions_prises.append(
                f"IP {incident.ip_source} ajoutée à la liste noire"
            )

        # Créer un tag Iceberg pour marquer l'état des tables au moment de l'incident
        for table in incident.tables_affectees:
            tag_name = f"incident_{incident.incident_id}
_{datetime.utcnow().strftime('%Y%m%d%H%M%S')}"
            self.spark.sql(f"""
                ALTER TABLE {table}
                CREATE TAG `{tag_name}`
            """)
            actions_prises.append(
                f"Tag de préservation créé pour {table}: {tag_name}"
            )

        return {
            "phase": "CONFINEMENT",
            "statut": "COMPLETE",
            "actions": actions_prises
        }

    def phase_3_investigation(self, incident: IncidentSecurite) -> dict:
        """
        Phase 3 : Investigation approfondie.

        Utilise les capacités du Lakehouse :
        - Time travel pour examiner l'état avant/après
        - Audit logs pour retracer les actions
        - Lignage pour identifier l'impact en aval
        """
        rapport_investigation = {
            "incident_id": incident.incident_id,
            "timestamp_debut_investigation": datetime.utcnow().isoformat()
        }

        # Analyser les accès de l'utilisateur suspect
        if incident.utilisateur_suspect:
            df_acces = self.spark.sql(f"""
                SELECT
                    timestamp,
                    action,
                    ressource_table,
                    requete_sql,
                    lignes_affectees,
                    ip_source
```

```
                FROM lakehouse.audit.evenements_unifies
                WHERE principal = '{incident.utilisateur_suspect}'
                AND timestamp >= '{(incident.timestamp_detection -
timedelta(days=30)).isoformat()}'
                ORDER BY timestamp DESC
            """)

            rapport_investigation["historique_acces"] = df_acces.collect()

        # Examiner les modifications sur les tables affectées
        for table in incident.tables_affectees:
            # Obtenir l'historique des snapshots
            df_snapshots = self.spark.sql(f"""
                SELECT * FROM {table}.history
                ORDER BY made_current_at DESC
            """)

            rapport_investigation[f"snapshots_{table}"] = df_snapshots.collect()

            # Comparer les données avant/après l'incident (si applicable)
            if incident.type_incident == "MODIFICATION":
                snapshot_avant = self._trouver_snapshot_avant(
                    table, incident.timestamp_detection
                )
                if snapshot_avant:
                    df_diff = self.spark.sql(f"""
                        SELECT 'SUPPRIME' as change_type, *
                        FROM {table} VERSION AS OF {snapshot_avant}
                        EXCEPT
                        SELECT 'SUPPRIME' as change_type, *
                        FROM {table}

                        UNION ALL

                        SELECT 'AJOUTE' as change_type, *
                        FROM {table}
                        EXCEPT
                        SELECT 'AJOUTE' as change_type, *
                        FROM {table} VERSION AS OF {snapshot_avant}
                    """)
                    rapport_investigation[f"modifications_{table}"] = df_diff.collect()

        # Identifier l'impact en aval via le lignage
        for table in incident.tables_affectees:
            tables_dependantes = self._obtenir_dependances_aval(table)
            rapport_investigation[f"impact_aval_{table}"] = tables_dependantes

        return rapport_investigation

    def phase_4_remediation(self, incident: IncidentSecurite,
                            rapport_investigation: dict) -> dict:
        """
        Phase 4 : Remédiation des dommages.

        Actions possibles :
        - Restauration depuis un snapshot antérieur
        - Correction des données modifiées
        - Renforcement des contrôles
        """
```

```
        actions_remediation = []

        # Si des données ont été modifiées, proposer une restauration
        if incident.type_incident == "MODIFICATION":
            for table in incident.tables_affectees:
                snapshot_propre = self._trouver_dernier_snapshot_propre(
                    table, incident.timestamp_detection
                )

                if snapshot_propre:
                    # Restauration (nécessite approbation)
                    self.notifier.demander_approbation(
                        destinataires=["proprietaire_donnees@entreprise.ca"],
                        sujet=f"Approbation restauration {table}",
                        corps=f"""
                        L'incident {incident.incident_id} a causé des modifications
                        non autorisées sur {table}.

                        Snapshot de restauration proposé: {snapshot_propre}

                        Veuillez approuver ou rejeter la restauration.
                        """
                    )
                    actions_remediation.append(
                        f"Demande de restauration envoyée pour {table}"
                    )

        # Renforcer les contrôles d'accès
        for table in incident.tables_affectees:
            # Activer l'audit détaillé si pas déjà actif
            self._activer_audit_detaille(table)
            actions_remediation.append(
                f"Audit détaillé activé pour {table}"
            )

        return {
            "phase": "REMEDIATION",
            "statut": "EN_COURS",
            "actions": actions_remediation
        }

    def phase_5_recuperation(self, incident: IncidentSecurite,
                             approbation_restauration: bool = False) -> dict:
        """
        Phase 5 : Récupération et retour à la normale.

        Actions :
        - Exécuter les restaurations approuvées
        - Rétablir les accès légitimes
        - Valider l'intégrité des données
        """
        actions_recuperation = []

        # Exécuter les restaurations si approuvées
        if approbation_restauration:
            for table in incident.tables_affectees:
                snapshot_propre = self._trouver_dernier_snapshot_propre(
                    table, incident.timestamp_detection
                )
```

```
                if snapshot_propre:
                    self.spark.sql(f"""
                        CALL system.rollback_to_snapshot(
                            table => '{table}',
                            snapshot_id => {snapshot_propre}
                        )
                    """)
                    actions_recuperation.append(
                        f"Restauration exécutée pour {table}"
                    )

        # Valider l'intégrité des données restaurées
        for table in incident.tables_affectees:
            validation = self._valider_integrite(table)
            actions_recuperation.append(
                f"Validation intégrité {table}: {validation}"
            )

        return {
            "phase": "RECUPERATION",
            "statut": "COMPLETE",
            "actions": actions_recuperation
        }

    def phase_6_lecons_apprises(self, incident: IncidentSecurite,
                                rapport_complet: dict) -> dict:
        """
        Phase 6 : Documentation et amélioration continue.

        Produit un rapport final incluant :
        - Chronologie complète
        - Cause racine
        - Actions correctives
        - Recommandations
        """
        rapport_final = {
            "incident_id": incident.incident_id,
            "date_cloture": datetime.utcnow().isoformat(),
            "chronologie": self._construire_chronologie(incident),
            "cause_racine": "",  # À compléter par l'analyste
            "donnees_exposees": [],  # À évaluer
            "actions_correctives_implementees": [],
            "recommandations": [
                "Renforcer la formation des utilisateurs",
                "Réviser les politiques d'accès",
                "Augmenter la fréquence des audits"
            ],
            "indicateurs_surveillance": []
        }

        # Stocker le rapport pour référence future
        self._archiver_rapport(rapport_final)

        # Notifier les parties prenantes de la clôture
        self.notifier.notification_cloture(
            destinataires=["securite@entreprise.ca",
                           "gouvernance@entreprise.ca"],
            rapport=rapport_final
```

```
        )

        return rapport_final
```

Ce framework de réponse aux incidents exploite les capacités uniques du Lakehouse Iceberg. Le time travel permet d'examiner l'état exact des données à tout moment dans le passé. Les tags préservent des points de référence pour l'investigation. L'audit unifié retrace chaque action effectuée. Le lignage identifie l'impact potentiel en cascade sur les systèmes consommateurs.

IV.13.6 Études de Cas Canadiennes

Étude de cas : Banque Nationale du Canada
Secteur : Services financiers
Défi : Moderniser l'architecture de données pour supporter l'analytique avancée et l'IA tout en maintenant la conformité BSIF, Loi 25 et SOX. L'infrastructure legacy Hadoop/Hive ne supportait pas les exigences de gouvernance granulaire.
Solution : Migration vers un Lakehouse Iceberg sur AWS avec :
- Apache Polaris comme catalogue centralisé avec RBAC et credential vending
- Unity Catalog (Databricks) pour les contrôles fins (RLS, column masking)
- OpenLineage/Marquez pour le lignage de bout en bout
- Pipeline automatisé de conformité Loi 25 (effacement, portabilité)
Architecture :
- 3 environnements isolés (dev, staging, prod) avec politiques RBAC distinctes
- Classification automatique des données via ML (détection PII)
- Chiffrement SSE-KMS avec rotation des clés
- Audit unifié CloudTrail + Polaris + Spark dans une table Iceberg dédiée
Résultats :
- Temps de réponse aux demandes d'accès BSIF réduit de 2 semaines à 4 heures
- 100 % des demandes Loi 25 traitées dans le délai légal de 30 jours
- Réduction de 60 % des coûts de conformité grâce à l'automatisation
- Zero incident de sécurité des données depuis la mise en production

Étude de cas : Hydro-Québec
Secteur : Énergie (société d'État)
Défi : Centraliser les données de 4,3 millions de clients et des réseaux de distribution tout en respectant les exigences de souveraineté des données et la Loi 25. Les données incluent des informations sensibles sur la consommation énergétique des résidences.
Solution : Lakehouse souverain déployé exclusivement dans les régions canadiennes :
- Stockage S3 dans ca-central-1 (Montréal) avec réplication vers ca-west-1 (Calgary)
- Nessie comme catalogue open source avec contrôles d'accès basés sur les branches
- Masquage dynamique des adresses et données de consommation pour les analystes
- Intégration avec le système de gestion des consentements pour la Loi 25
Architecture :
- Séparation physique des données clients (zone restreinte) et données opérationnelles (zone standard)

- Anonymisation automatique des données pour l'analytique générale
- Vues sécurisées pour chaque équipe avec périmètre géographique
- Audit complet avec rétention 10 ans pour conformité Loi sur les archives

Résultats :
- Conformité Loi 25 certifiée par audit externe
- Temps d'extraction pour demandes d'accès réduit de 5 jours à 2 heures
- Démocratisation des données : 200+ analystes avec accès sécurisé (vs 15 auparavant)
- Économies de 2,1 M$ CAD/an en licences et infrastructure

Étude de cas : Shopify

Secteur : Commerce électronique (technologie)

Défi : Gérer les données de millions de marchands et leurs clients à travers le monde, avec conformité multi-juridictionnelle (RGPD, CCPA, Loi 25, LPRPDE). Volume de données massif nécessitant des contrôles d'accès performants.

Solution : Lakehouse global avec gouvernance fédérée :
- Tables Iceberg partitionnées par région géographique pour isolation réglementaire
- Attribute-Based Access Control (ABAC) basé sur les tags de classification
- Pipeline automatisé de « Right to Be Forgotten » traitant 50 000+ demandes/mois
- Système de consentement intégré au pipeline d'ingestion

Architecture :
- Multi-région avec résidence des données (Canada pour marchands canadiens)
- Tags automatiques : PII, FINANCIAL, MERCHANT_CONFIDENTIAL
- Politiques ABAC appliquées à la découverte des tags
- Effacement distribué avec certificats de conformité générés automatiquement

Résultats :
- SLA de 72h pour les demandes d'effacement (vs 30 jours légaux)
- Traitement de 50 000+ demandes de droits/mois entièrement automatisé
- Audit de conformité RGPD/Loi 25 passé sans recommandation majeure
- Réduction de 80 % de l'équipe dédiée à la conformité manuelle

IV.13.7 Résumé

Ce chapitre a présenté une approche holistique de la sécurité, de la gouvernance et de la conformité du Data Lakehouse. Les points essentiels à retenir sont structurés ci-dessous.

Architecture de Sécurité Multicouche

La sécurité du Lakehouse repose sur quatre couches complémentaires. Le **stockage objet** assure le chiffrement au repos et en transit, l'isolation réseau et les politiques d'accès de base. Le **format de table Iceberg** permet le chiffrement natif des fichiers Parquet avec contrôle par colonne. Le **catalogue** (Polaris, Unity Catalog, Nessie) centralise le RBAC et le credential vending pour une gouvernance cohérente. Les **moteurs de calcul** implémentent les contrôles fins (RLS, masquage) qui nécessitent l'accès aux données.

La séparation des responsabilités entre ces couches permet une défense en profondeur. Une compromission à un niveau est contenue par les contrôles des autres niveaux.

Contrôles d'Accès Granulaires

Le RBAC au niveau du catalogue constitue le fondement, contrôlant l'accès aux objets (tables, namespaces). Apache Polaris et Unity Catalog offrent des implémentations matures avec credential vending, éliminant le besoin de distribuer des identifiants permanents.

Le Row-Level Security filtre automatiquement les données visibles selon l'identité de l'utilisateur. Cette approche permet de partager une même table entre équipes avec des périmètres de données différents.

Le Column Masking protège les colonnes sensibles en affichant des valeurs masquées aux utilisateurs non autorisés. Plusieurs patterns de masquage (préservation de format, tokenisation, généralisation) permettent de préserver l'utilité analytique tout en protégeant la confidentialité.

L'ABAC à grande échelle permet de définir des politiques basées sur des tags plutôt que sur des objets individuels. Cette approche simplifie considérablement la gestion pour les organisations avec des milliers de tables.

Gouvernance des Données

OpenLineage s'impose comme le standard pour la collecte du lignage à travers les outils et plateformes. L'intégration avec Spark, Flink et Airflow permet de capturer automatiquement les transformations et leurs dépendances.

La qualité des données doit être validée automatiquement à chaque étape du pipeline. Les frameworks comme Great Expectations permettent de définir des « attentes » déclaratives qui sont vérifiées à chaque exécution.

Le catalogage métier complète le catalogue technique en documentant le contexte, la propriété et la classification des données. Cette documentation est essentielle pour la transparence requise par les réglementations.

Conformité Canadienne

La Loi 25 du Québec impose des exigences strictes de consentement explicite, droit à l'effacement et portabilité. Ces exigences nécessitent des pipelines automatisés et une gestion rigoureuse des snapshots Iceberg. L'expiration des snapshots après suppression garantit l'effectivité du droit à l'effacement.

La LPRPDE fédérale établit les dix principes fondamentaux que l'architecture doit supporter. Bien que moins stricte que la Loi 25, elle s'applique à toutes les organisations canadiennes dans le secteur privé.

Les directives BSIF (B-13, E-21) ajoutent des exigences de résilience et de gestion des tiers pour les institutions financières. La classification des données selon leur criticité et leur sensibilité guide les contrôles à appliquer.

Audit et Réponse aux Incidents

La journalisation des accès à plusieurs niveaux (stockage, catalogue, calcul) permet de reconstituer l'activité complète lors d'investigations. L'unification de ces journaux dans une table Iceberg dédiée simplifie l'analyse.

Les alertes proactives détectent les comportements anormaux avant qu'ils ne causent des dommages significatifs. Les patterns à surveiller incluent les accès hors heures normales, les volumes anormaux et les tentatives répétées refusées.

Le framework de réponse aux incidents exploite les capacités uniques du Lakehouse. Le time travel permet d'examiner l'état des données à tout moment. Les tags préservent des points de référence pour l'investigation. La restauration depuis un snapshot antérieur permet de remédier aux modifications non autorisées.

Recommandations Stratégiques

Pour les organisations qui construisent ou modernisent leur Lakehouse, les recommandations suivantes guident l'approche :

1. **Adopter le credential vending** via un catalogue moderne (Polaris, Unity Catalog) pour éliminer les identifiants permanents distribués aux moteurs de calcul
2. **Implémenter la classification automatique** des données dès l'ingestion pour appliquer les contrôles appropriés sans intervention manuelle
3. **Automatiser les processus de conformité** (effacement, portabilité, audit) pour respecter les délais légaux et réduire les coûts opérationnels
4. **Unifier les journaux d'audit** des différentes couches (stockage, catalogue, calcul) dans une table Iceberg dédiée pour les investigations
5. **Documenter les métadonnées métier** dans un catalogue accessible pour permettre la découverte et démontrer la transparence requise par les réglementations
6. **Expirer proactivement les snapshots** contenant des données supprimées pour garantir l'effectivité du droit à l'effacement
7. **Déployer dans les régions canadiennes** (ca-central-1, ca-west-1 pour AWS; Canada Central, Canada East pour Azure) pour assurer la souveraineté des données et simplifier la conformité
8. **Établir un plan de réponse aux incidents** documenté et testé, exploitant les capacités de time travel et d'audit du Lakehouse
9. **Former les équipes** aux bonnes pratiques de sécurité et aux exigences réglementaires spécifiques à leur rôle
10. **Réviser périodiquement** les politiques d'accès et les classifications de données pour s'assurer qu'elles restent alignées avec les besoins métier et les exigences réglementaires

La sécurité et la gouvernance ne sont pas des projets ponctuels mais des pratiques continues. L'évolution constante des réglementations (modernisation de la LPRPDE, nouvelles directives BSIF) et des menaces de sécurité exige une architecture adaptable et une surveillance vigilante. Le Lakehouse, par sa nature ouverte et sa traçabilité inhérente via les snapshots Iceberg, offre une fondation solide pour cette gouvernance évolutive.

Références

1. Apache Polaris (2025). *Documentation officielle*. polaris.apache.org
2. Atlan (2025). *Apache Iceberg Tables Governance: A Practical Guide*. Documentation technique.
3. Dremio (2025). *Securing Your Apache Iceberg Data Lakehouse*. Blog technique.
4. Dremio (2025). *Credential Vending with Iceberg REST Catalogs*. Blog technique.

5. OpenLineage (2025). *An Open Standard for Data Lineage*. Documentation officielle.
6. Databricks (2025). *Row Filters and Column Masks in Unity Catalog*. Documentation technique.
7. AWS (2025). *Governance and Access Control for Apache Iceberg on AWS*. Prescriptive Guidance.
8. Commission d'accès à l'information du Québec (2024). *Loi 25 - Guide de mise en œuvre*. Publication officielle.
9. Bureau du surintendant des institutions financières (2024). *Ligne directrice B-13 : Gestion du risque lié aux technologies*. Directive réglementaire.
10. Office of the Privacy Commissioner of Canada (2024). *PIPEDA in Brief*. Guide de conformité.

Chapitre IV.14 - L'Intégration avec Microsoft Fabric et Power BI

Introduction

L'écosystème des données d'entreprise a longtemps souffert d'une fragmentation qui complique la gouvernance, multiplie les coûts et ralentit les initiatives analytiques. Les organisations canadiennes, comme leurs homologues internationales, se retrouvent souvent avec des données éparpillées entre différentes plateformes, formats et régions infonuagiques. Dans ce contexte, l'émergence de Microsoft Fabric représente une évolution majeure : une plateforme unifiée de données et d'analytique qui place OneLake — un lac de données logique unifié — au cœur de son architecture.

Ce chapitre explore l'intégration entre Apache Iceberg et l'écosystème Microsoft Fabric, une convergence qui illustre parfaitement la tendance vers l'interopérabilité des formats de table ouverts. Alors que Fabric utilise Delta Lake comme format natif, la plateforme a développé des capacités sophistiquées de virtualisation des métadonnées qui permettent de travailler de manière transparente avec les tables Iceberg, sans duplication de données ni migration complexe.

L'enjeu est considérable pour les architectes de données : comment tirer parti des investissements existants en Apache Iceberg tout en bénéficiant de l'intégration native de Fabric avec Power BI, Microsoft 365 et l'écosystème Azure? Comment exploiter le mode Direct Lake de Power BI pour obtenir des performances analytiques optimales sur des données Lakehouse? Ce chapitre répond à ces questions en détaillant les mécanismes de virtualisation OneLake, les stratégies d'intégration et les considérations de performance pour les architectures hybrides.

La pertinence de cette intégration pour le contexte canadien est particulière. Les organisations canadiennes opèrent souvent dans un environnement multi-infonuagique, avec des exigences strictes de résidence des données. La capacité de Fabric à unifier des données provenant de diverses sources — y compris des tables Iceberg hébergées sur AWS S3 ou Google Cloud Storage — tout en maintenant une gouvernance centralisée dans les régions Azure canadiennes, répond directement à ces besoins. Avec plus de 28 000 organisations ayant adopté Fabric à l'échelle mondiale, dont plusieurs entreprises canadiennes de premier plan, cette intégration devient un élément central de l'architecture de données moderne.

IV.14.1 OneLake Shortcuts et Virtualisation

L'Architecture OneLake : Le Lac de Données Unifié

OneLake constitue le fondement architectural de Microsoft Fabric. Contrairement aux approches traditionnelles où chaque charge de travail analytique dispose de son propre stockage isolé, OneLake offre un

lac de données logique unique pour l'ensemble de l'organisation. Cette architecture s'inspire du modèle OneDrive pour les données d'entreprise : un espace de stockage unifié où toutes les données analytiques coexistent, accessibles par l'ensemble des moteurs de calcul Fabric.

OneLake est construit sur Azure Data Lake Storage (ADLS) Gen2, ce qui garantit la compatibilité avec les APIs et SDKs ADLS existants. Toutes les charges de travail Fabric — entrepôts de données, lakehouses, pipelines de données, modèles sémantiques Power BI — stockent automatiquement leurs données dans OneLake au format Delta Parquet. Cette standardisation sur un format ouvert élimine les silos de données et simplifie considérablement la gouvernance.

L'organisation hiérarchique d'OneLake suit une structure logique. Au niveau supérieur, le locataire (tenant) établit les politiques de sécurité, de conformité et de gestion des données applicables à l'ensemble de l'organisation. Les espaces de travail (workspaces) permettent ensuite de distribuer la propriété et les politiques d'accès entre différentes équipes ou unités d'affaires. Chaque espace de travail est associé à une capacité Fabric liée à une région spécifique et facturée séparément. À l'intérieur d'un espace de travail, les éléments de données — lakehouses, entrepôts, bases de données — représentent les conteneurs logiques pour les tables et fichiers.

Cette architecture présente plusieurs avantages stratégiques pour les organisations canadiennes. Premièrement, la gouvernance centralisée au niveau du locataire permet d'appliquer uniformément les politiques de conformité requises par les réglementations canadiennes comme LPRPDE. Deuxièmement, la flexibilité des espaces de travail autorise une décentralisation de la propriété des données conforme aux principes du Data Mesh, tout en maintenant des standards organisationnels cohérents. Troisièmement, l'association des capacités à des régions Azure canadiennes (Canada Central, Canada East) garantit la résidence des données sur le territoire canadien.

Les Raccourcis OneLake : Virtualisation sans Duplication

Les raccourcis (shortcuts) OneLake représentent l'innovation clé qui permet l'intégration avec Apache Iceberg. Un raccourci est une référence vers des données stockées dans d'autres emplacements — que ce soit à l'intérieur d'OneLake, dans d'autres espaces de travail, ou dans des systèmes de stockage externes comme ADLS, Amazon S3, Google Cloud Storage, ou même des sources sur site.

Le principe fondamental des raccourcis est la virtualisation sans mouvement de données. Lorsqu'un raccourci est créé, les fichiers et dossiers référencés apparaissent comme s'ils étaient stockés localement dans OneLake, mais aucune copie physique n'est effectuée. Cette approche offre plusieurs bénéfices majeurs :

Élimination de la duplication des données. Les organisations peuvent accéder à des téraoctets de données stockées dans diverses sources sans multiplier les coûts de stockage. Une table Iceberg de 500 Go stockée sur S3 peut être accessible depuis Fabric sans consommer 500 Go supplémentaires dans OneLake.

Gouvernance unifiée. Malgré la distribution physique des données, OneLake applique de manière uniforme les politiques de sécurité et de gouvernance. Les contrôles d'accès définis dans Fabric s'appliquent également aux données accédées via raccourcis.

Fraîcheur des données. Contrairement aux approches ETL traditionnelles qui introduisent une latence entre la source et la cible, les raccourcis donnent accès aux données les plus récentes. Toute mise à jour dans la source est immédiatement reflétée via le raccourci.

Flexibilité multi-infonuagique. Les raccourcis supportent une variété de sources : Azure Data Lake Storage Gen2, Amazon S3, Google Cloud Storage, stockage compatible S3, Dataverse, et même des sources sur site via les passerelles de données. Cette flexibilité est particulièrement précieuse pour les organisations canadiennes opérant dans des environnements hybrides ou multi-infonuagiques.

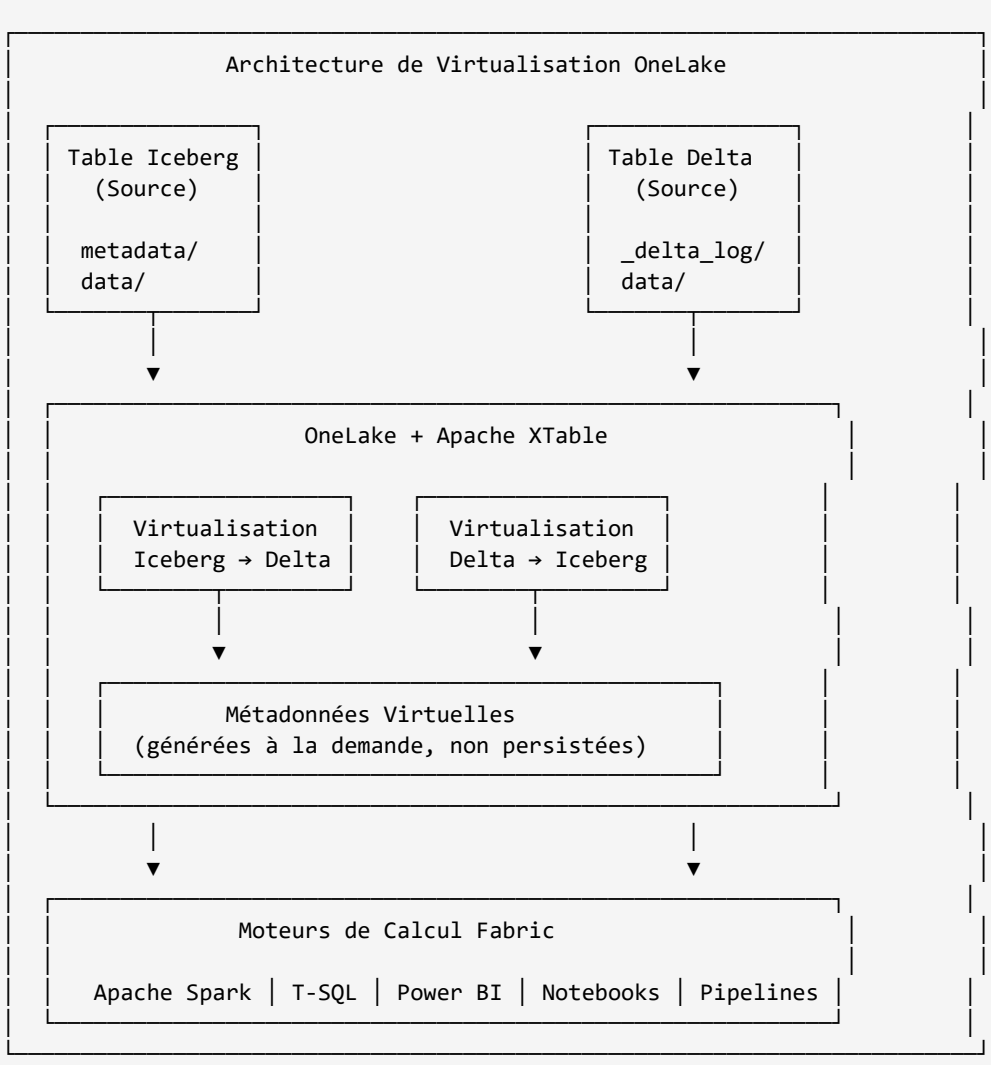
La Virtualisation des Métadonnées : Le Pont Iceberg-Delta

La véritable innovation technique de Fabric réside dans sa capacité de virtualisation des métadonnées entre les formats de table. Cette fonctionnalité permet aux tables Iceberg d'être interprétées comme des tables Delta Lake, et vice versa, sans aucune conversion ou duplication des fichiers de données sous-jacents.

Lorsqu'un raccourci est créé vers un dossier contenant une table Iceberg, OneLake génère automatiquement les métadonnées Delta Lake correspondantes (le Delta Log) pour cette table. Cette génération est transparente et dynamique : lorsque des mises à jour sont effectuées sur la table Iceberg source, les métadonnées Delta fraîches sont servies via le raccourci lors des requêtes subséquentes.

En coulisses, cette fonctionnalité utilise Apache XTable (incubating) pour la conversion des métadonnées entre formats. XTable est un projet open source soutenu par Microsoft et d'autres acteurs majeurs comme Google, Snowflake et Databricks. Son approche est omni-directionnelle : il peut traduire les métadonnées entre Delta Lake, Apache Iceberg et Apache Hudi, tout en maintenant une seule copie des fichiers de données Parquet sous-jacents.

Microsoft a également enrichi les fonctionnalités de XTable pour Fabric. Par exemple, la conversion des vecteurs de suppression Delta (deletion vectors) en fichiers de suppression positionnelle Iceberg garantit une fidélité complète dans la traduction des opérations de mutation.



Configuration de l'Intégration Iceberg vers Fabric

L'intégration d'une table Iceberg existante dans Microsoft Fabric s'effectue en quelques étapes. Le processus varie légèrement selon que la table Iceberg est stockée dans un système de stockage externe ou directement dans OneLake.

Scénario 1 : Table Iceberg dans un stockage externe (S3, ADLS, GCS)

Pour une table Iceberg stockée dans Amazon S3, Azure Data Lake Storage ou Google Cloud Storage, le processus implique la création d'un raccourci vers le dossier de la table. Voici les étapes détaillées :

1. **Préparation des accès.** Configurer les informations d'authentification pour le stockage externe. Pour S3, cela implique un ARN de rôle IAM ou des clés d'accès. Pour ADLS, un principal de service ou une identité managée.
2. **Navigation vers le Lakehouse.** Dans l'espace de travail Fabric, ouvrir le Lakehouse qui servira de point d'accès aux données Iceberg.
3. **Création du raccourci.** Depuis la section Tables du Lakehouse, sélectionner « Nouveau raccourci » puis choisir le type de source (S3, ADLS, GCS, etc.).
4. **Configuration de la connexion.** Entrer les informations de connexion et naviguer jusqu'au dossier contenant la table Iceberg. Sélectionner uniquement le dossier de niveau table — ne pas sélectionner les sous-dossiers « data » ou « metadata ».
5. **Validation de la conversion.** Une fois le raccourci créé, OneLake effectue automatiquement la virtualisation. La table apparaît comme une table Delta dans le Lakehouse. Un dossier virtuel `_delta_log/` est généré, contenant les métadonnées Delta et un fichier `latest_conversion_log.txt` indiquant le statut de la conversion.

Scénario 2 : Écriture de tables Iceberg directement dans OneLake

Cette approche est particulièrement pertinente pour les organisations utilisant Snowflake comme moteur d'écriture Iceberg. Depuis l'annonce du partenariat Microsoft-Snowflake, les utilisateurs Snowflake sur Azure peuvent écrire des tables Iceberg directement dans OneLake.

La configuration nécessite la création d'un volume externe Snowflake pointant vers OneLake :

```
-- Création du volume externe dans Snowflake
CREATE OR REPLACE EXTERNAL VOLUME FabricExVol
STORAGE_LOCATIONS = (
  (
    NAME = 'FabricExVol'
    STORAGE_PROVIDER = 'AZURE'
    STORAGE_BASE_URL = 'azure://onelake.dfs.fabric.microsoft.com/
                        <NomEspaceTravail>/<NomLakehouse>.Lakehouse/Files/'
    AZURE_TENANT_ID = '<ID_Locataire_Azure>'
  )
);

-- Création d'un catalogue Iceberg
CREATE OR REPLACE ICEBERG CATALOG FabricCatalog
EXTERNAL_VOLUME = 'FabricExVol'
CATALOG_TYPE = 'ICEBERG_DIRECTORY';

-- Création d'une table Iceberg dans OneLake
CREATE ICEBERG TABLE ma_table_iceberg (
  id INTEGER,
```

```
  nom VARCHAR,
  date_creation TIMESTAMP
)
CATALOG = 'FabricCatalog'
EXTERNAL_VOLUME = 'FabricExVol'
BASE_LOCATION = 'tables/ma_table';
```

Une fois la table créée, un raccourci OneLake peut être configuré pour la rendre accessible aux moteurs Fabric.

Migration
De : Tables Iceberg sur stockage externe (S3, ADLS)
Vers : Intégration Fabric via raccourcis OneLake
Stratégie : Création de raccourcis sans mouvement de données. Les tables restent physiquement dans leur emplacement d'origine. La virtualisation des métadonnées permet leur utilisation native dans Fabric. Aucune modification des pipelines d'écriture existants n'est requise.

La Conversion Delta vers Iceberg : Bidirectionnalité Complète

Fabric ne se limite pas à l'importation de tables Iceberg. La plateforme offre également la conversion inverse : les tables Delta Lake natives de Fabric peuvent être exposées automatiquement au format Iceberg pour être consommées par des moteurs externes comme Trino, Dremio ou Snowflake.

Cette fonctionnalité, annoncée en 2025, complète la boucle d'interopérabilité. Les organisations peuvent désormais :

- **Écrire dans Fabric** : Les pipelines Data Factory, notebooks Spark ou flux Dataflow Gen2 créent des tables Delta dans OneLake.
- **Lire depuis n'importe quel moteur Iceberg** : Ces tables sont automatiquement accessibles au format Iceberg, sans configuration supplémentaire.

Pour activer cette conversion, il suffit d'activer le paramètre délégué OneLake « Enable Delta Lake to Apache Iceberg table format virtualization » dans les paramètres de l'espace de travail. Une fois activé, toutes les tables Delta de l'espace de travail deviennent automatiquement lisibles via des lecteurs Iceberg.

La vérification de la conversion réussie s'effectue en examinant le répertoire de la table. Un dossier `metadata/` contenant les fichiers de métadonnées Iceberg apparaît, ainsi qu'un fichier de journal de conversion indiquant l'horodatage de la dernière conversion et les éventuelles erreurs.

Étude de cas : Société d'État canadienne
Secteur : Services publics
Défi : Unifier les analyses entre un entrepôt Snowflake existant et les nouveaux rapports Power BI, tout en respectant les exigences de résidence des données canadiennes.
Solution : Déploiement de Fabric avec les données stockées dans la région Canada Central. Les tables Iceberg écrites par Snowflake sont virtualisées comme Delta pour Power BI. Les nouvelles tables créées dans Fabric sont exposées comme Iceberg pour les analyses Snowflake existantes.
Résultats : Élimination de 3 pipelines ETL de synchronisation, réduction de 40% des coûts de stockage par l'élimination de la duplication, conformité maintenue avec LPRPDE.

Types de Raccourcis et Configurations Avancées

OneLake supporte plusieurs types de raccourcis, chacun adapté à des scénarios spécifiques.

Raccourcis OneLake internes. Ces raccourcis pointent vers des données situées dans d'autres espaces de travail Fabric au sein du même locataire. Ils sont particulièrement utiles pour implémenter une architecture Data Mesh où chaque domaine métier possède son propre Lakehouse, mais partage certaines tables avec d'autres domaines. Les performances sont optimales puisque les données restent dans OneLake.

Raccourcis Azure Data Lake Storage Gen2. Pour les organisations ayant des investissements existants dans ADLS, ces raccourcis permettent d'intégrer les données sans migration. L'authentification peut utiliser des clés de compte, des signatures d'accès partagé (SAS), des principaux de service, ou des identités managées. Pour les données sensibles, les identités managées offrent la meilleure posture de sécurité en évitant la gestion de secrets.

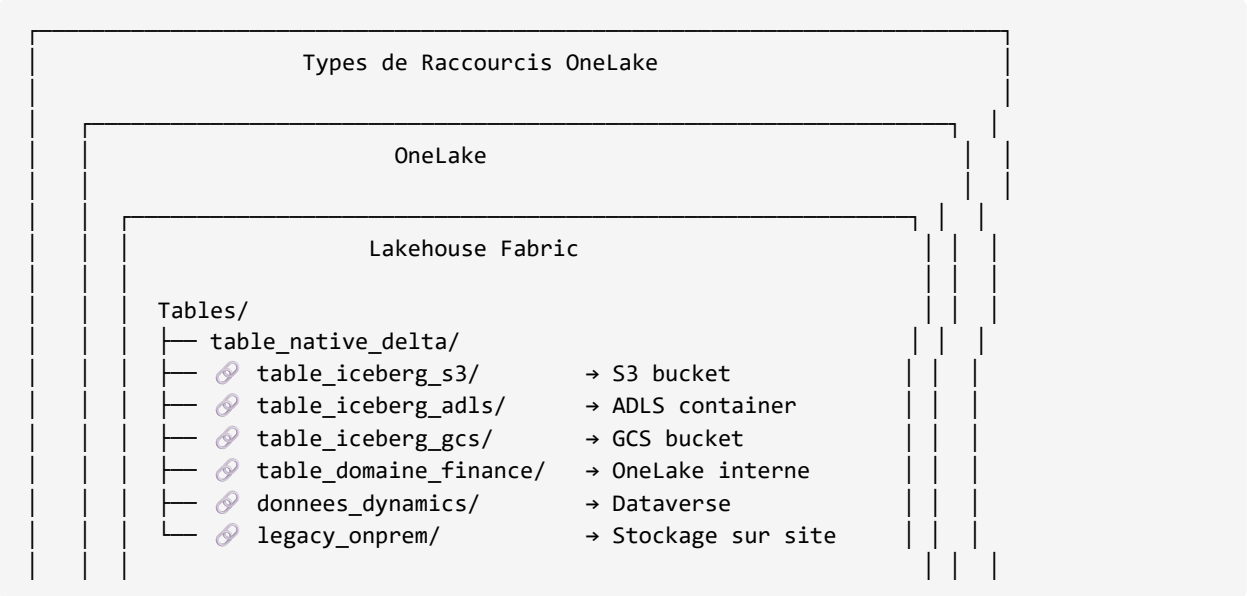
Raccourcis Amazon S3. Ces raccourcis supportent les tables Iceberg stockées dans l'écosystème AWS. L'authentification utilise soit des clés d'accès IAM, soit des rôles IAM assumés. Pour les charges de travail de production, les rôles assumés avec des politiques de confiance appropriées sont recommandés. La région S3 impacte la latence — idéalement, choisir une région AWS proche de la région Azure hébergeant la capacité Fabric.

Raccourcis Google Cloud Storage. Similaires aux raccourcis S3, ils permettent d'accéder aux données dans GCP. L'authentification utilise des comptes de service GCP avec les permissions appropriées sur les buckets.

Raccourcis stockage compatible S3. De nombreux systèmes de stockage implémentent l'API S3, incluant MinIO, Cloudflare R2, DigitalOcean Spaces, et d'autres. Ces raccourcis permettent d'intégrer ces sources dans OneLake.

Raccourcis Dataverse. Pour les organisations utilisant Microsoft Dataverse (Power Platform, Dynamics 365), ces raccourcis permettent d'intégrer les données transactionnelles dans le Lakehouse pour l'analytique.

Raccourcis vers sources sur site. Annoncés en 2024 et en disponibilité générale en 2025, ces raccourcis permettent d'accéder aux données derrière des pare-feux ou dans des réseaux privés virtuels via les passerelles de données locales. Cette fonctionnalité est particulièrement précieuse pour les organisations canadiennes ayant des exigences de résidence de données sur site.



```
│   │  └──────────────────────────────────────────────────────────┘ │  │
│   │                                                              │  │
│   │  Toutes les tables apparaissent comme Delta Lake natif       │  │
│   │  Virtualisation automatique des métadonnées Iceberg          │  │
│   └──────────────────────────────────────────────────────────────┘  │
└─────────────────────────────────────────────────────────────────────┘
```

Configuration de l'Authentification pour les Raccourcis Externes

La configuration sécurisée de l'authentification est cruciale pour les raccourcis vers des stockages externes. Voici les meilleures pratiques pour chaque type de source.

Pour Amazon S3 avec rôle IAM assumé :

```
{
  "roleName": "FabricOneLakeAccess",
  "trustPolicy": {
    "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Principal": {
          "AWS": "arn:aws:iam::FABRIC_AWS_ACCOUNT:root"
        },
        "Action": "sts:AssumeRole",
        "Condition": {
          "StringEquals": {
            "sts:ExternalId": "VOTRE_EXTERNAL_ID_UNIQUE"
          }
        }
      }
    ]
  },
  "permissions": {
    "s3:GetObject": "arn:aws:s3:::votre-bucket/tables/*",
    "s3:ListBucket": "arn:aws:s3:::votre-bucket"
  }
}
```

Pour ADLS avec identité managée :

L'approche recommandée utilise une identité managée assignée à l'espace de travail Fabric. Cette identité reçoit ensuite les permissions appropriées sur le compte de stockage ADLS via des attributions de rôle Azure RBAC :

- **Storage Blob Data Reader** : Pour l'accès en lecture seule
- **Storage Blob Data Contributor** : Si l'écriture via Fabric est également requise

Cette approche élimine la nécessité de gérer des secrets et s'intègre naturellement avec les pratiques de sécurité Azure des organisations canadiennes.

Considérations de Performance pour les Raccourcis

L'utilisation de raccourcis OneLake pour accéder aux tables Iceberg introduit certaines considérations de performance que les architectes doivent comprendre.

Latence réseau. Lorsque les données sont stockées dans un système externe (S3, GCS), chaque requête traverse le réseau vers ce système. La latence dépend de la proximité géographique et de la bande passante disponible. Pour les charges de travail sensibles à la latence, stocker les données directement dans OneLake (région Azure proche des utilisateurs) offre de meilleures performances.

Coûts de sortie (egress). L'accès aux données via raccourcis depuis des fournisseurs infonuagiques tiers (AWS, GCP) génère des frais de sortie de données. Ces coûts peuvent devenir significatifs pour des charges de travail analytiques à haut volume. Une analyse coût-bénéfice est recommandée pour déterminer si la duplication des données dans OneLake serait plus économique.

Fraîcheur des métadonnées. La virtualisation des métadonnées Iceberg vers Delta s'effectue lors de l'accès. Pour les tables Iceberg fréquemment mises à jour, cette conversion peut introduire une légère latence lors de la première requête après une mise à jour. Les requêtes subséquentes bénéficient des métadonnées mises en cache.

Limites temporaires. Certaines fonctionnalités sont encore en prévisualisation. Par exemple, tous les types de données Iceberg ne sont pas encore supportés pour la conversion. Les architectes doivent consulter la documentation Microsoft pour les limitations actuelles.

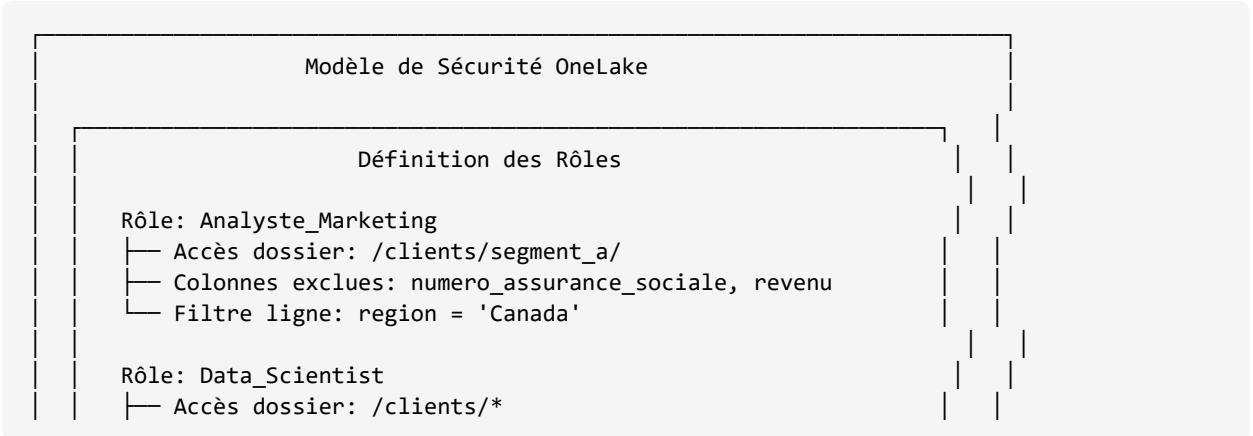
> **Performance**
> Pour les tables Iceberg accédées fréquemment via raccourcis OneLake, considérer le stockage direct dans OneLake si le volume de données et la fréquence d'accès justifient les coûts de migration. La règle empirique : si les frais de sortie mensuels dépassent le coût de stockage OneLake équivalent, la migration est économiquement avantageuse.

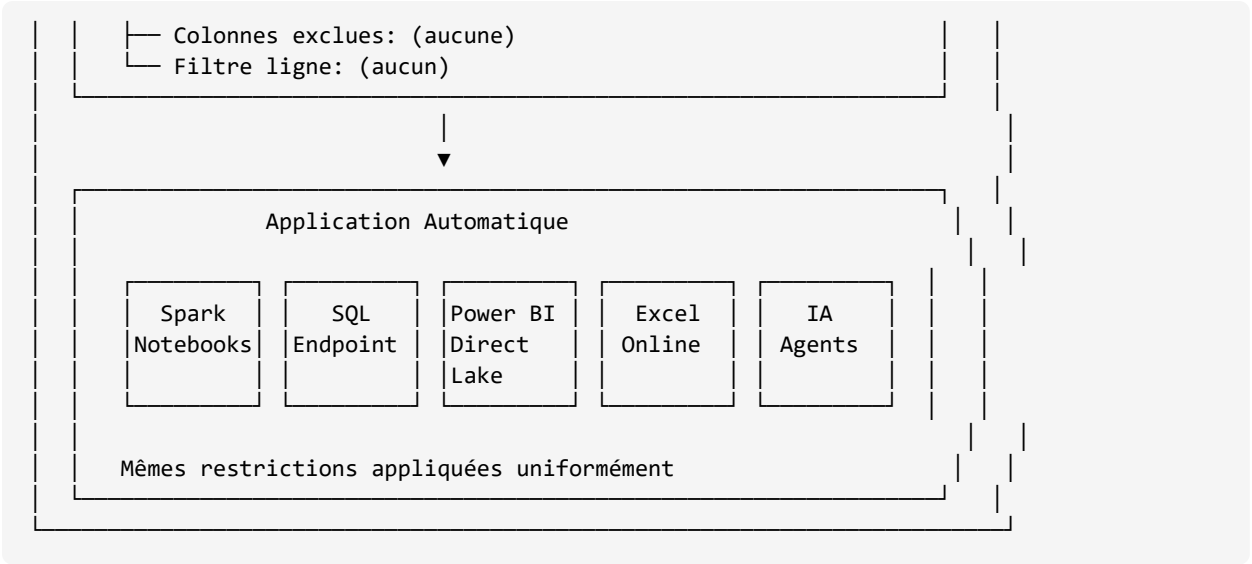
Sécurité OneLake : Gouvernance Unifiée

La sécurité OneLake, disponible en prévisualisation publique depuis 2025, introduit un modèle de contrôle d'accès fin qui s'applique uniformément à toutes les données, y compris celles accédées via raccourcis vers des tables Iceberg.

Ce modèle permet de définir des rôles de sécurité avec des permissions au niveau des dossiers, des lignes et des colonnes. Une fois définie, cette sécurité se propage automatiquement à travers tous les moteurs de calcul Fabric : notebooks Spark, points de terminaison SQL, Power BI, et même les agents de données IA.

Pour les tables Iceberg intégrées via raccourcis, cette sécurité offre une couche de gouvernance supplémentaire. Même si la table Iceberg source ne dispose pas de contrôles d'accès granulaires, OneLake peut appliquer des restrictions sur les données exposées aux utilisateurs Fabric.





L'intégration de la sécurité OneLake avec les raccourcis Iceberg permet aux propriétaires de données de démocratiser l'accès tout en maintenant le contrôle. Les données peuvent être partagées via raccourcis depuis le Lakehouse d'un analyste métier, mais les restrictions de sécurité définies par le propriétaire original continuent de s'appliquer.

IV.14.2 Power BI Direct Lake : Latence et Performance

Comprendre les Modes de Stockage Power BI

Avant d'explorer le mode Direct Lake, il est essentiel de comprendre les modes de stockage traditionnels de Power BI et leurs compromis respectifs.

Mode Import. Dans ce mode, les données sont copiées dans le modèle Power BI et stockées dans des fichiers propriétaires .idf utilisant le moteur de compression VertiPaq. Ce mode offre les meilleures performances de requête grâce au stockage en mémoire et à la compression columaire optimisée. Cependant, il nécessite des actualisations périodiques pour refléter les changements dans les données sources, et la duplication des données augmente les coûts de stockage et les temps d'actualisation.

Mode DirectQuery. Ce mode interroge directement la source de données à chaque interaction utilisateur, garantissant ainsi la fraîcheur des données. Les performances dépendent toutefois fortement de la capacité et de l'optimisation du système source. Les requêtes complexes ou les grands volumes de données peuvent entraîner des temps de réponse inacceptables pour l'expérience utilisateur interactive.

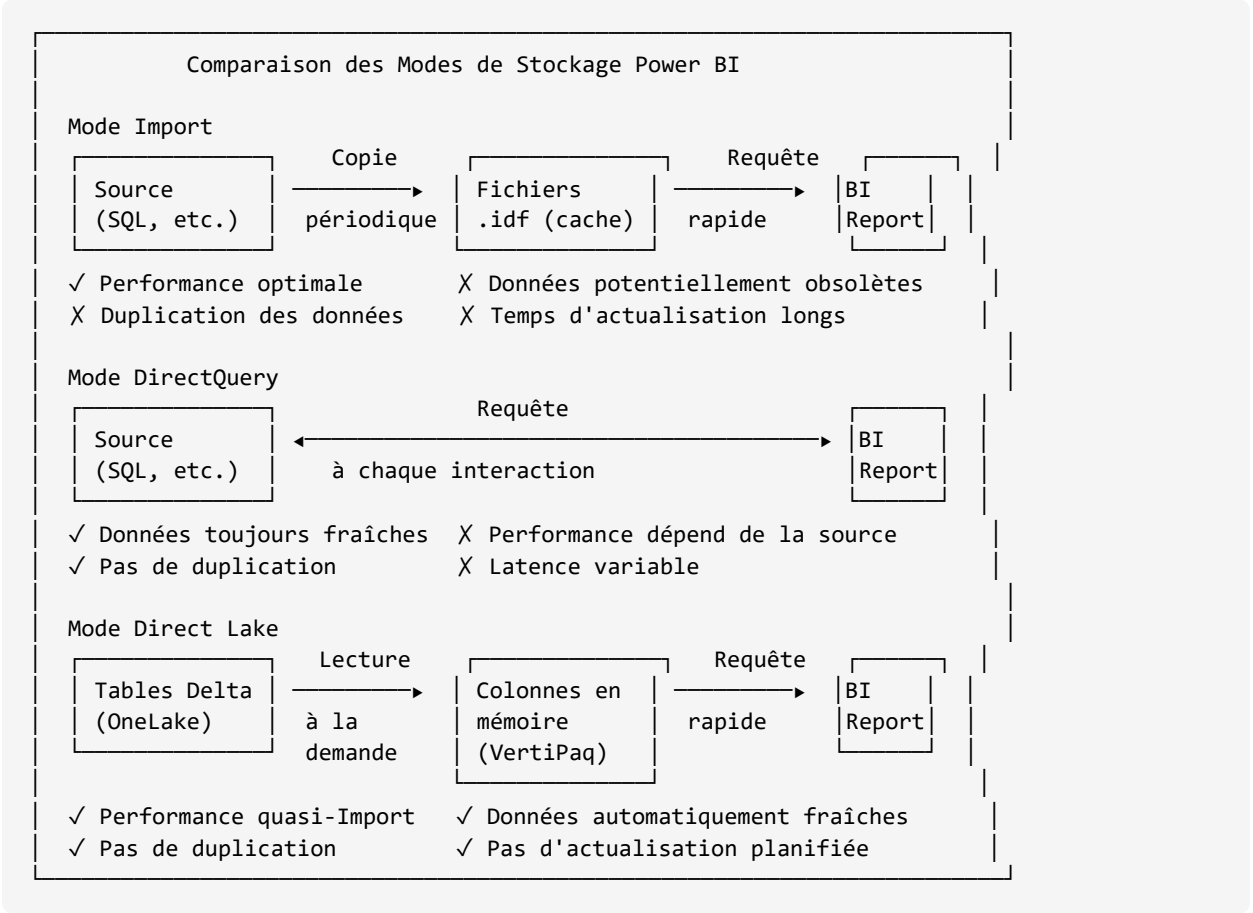
Mode Live Connection. Ce mode se connecte à un modèle sémantique Power BI ou Analysis Services existant, exploitant son cache en mémoire tout en maintenant une connexion active à la source. Il est limité aux sources Power BI ou Analysis Services.

Chacun de ces modes force un compromis entre performance et fraîcheur des données. Le mode Direct Lake, exclusif à Microsoft Fabric, brise ce compromis en offrant simultanément la vitesse du mode Import et la fraîcheur du mode DirectQuery.

L'Architecture Direct Lake

Direct Lake est un mode de stockage pour les tables d'un modèle sémantique Power BI qui élimine la nécessité d'importer ou de dupliquer les données. Au lieu de cela, il accède directement aux tables Delta stockées dans OneLake, permettant des analyses en temps quasi réel.

Le principe fondamental est simple : puisque les fichiers Parquet (format de stockage de Delta Lake) utilisent un stockage columnaire similaire aux fichiers .idf de VertiPaq, Power BI peut lire directement ces fichiers et charger les colonnes nécessaires en mémoire à la demande. Aucune copie préalable n'est nécessaire, et les changements dans les tables Delta sont automatiquement détectés et reflétés.



Le Mécanisme de Chargement à la Demande

Direct Lake utilise un mécanisme sophistiqué de chargement à la demande basé sur la « température » des colonnes. Plutôt que de précharger toutes les données en mémoire, le système charge uniquement les colonnes nécessaires pour satisfaire chaque requête.

Chargement initial. Lors de la première ouverture d'un rapport Direct Lake, les colonnes requises par les visuels affichés sont chargées des fichiers Delta Parquet vers la mémoire VertiPaq. Cette opération initiale peut prendre quelques secondes selon le volume de données.

Cache basé sur la température. Le système attribue une « température » à chaque colonne basée sur la fréquence et la récence de son utilisation. Les colonnes fréquemment interrogées maintiennent une température élevée et restent en mémoire. Les colonnes peu utilisées ont une température basse et peuvent être évincées du cache si la mémoire devient limitée.

Transcodage à la volée. Delta Lake utilise une compression différente de VertiPaq. Lorsque les données sont chargées, elles sont « transcodées » à la volée vers un format que le moteur Analysis Services peut traiter efficacement.

Détection automatique des changements. Lorsque les tables Delta sous-jacentes sont mises à jour (nouvelles données ingérées, modifications), Direct Lake détecte automatiquement ces changements. Lors de la prochaine requête, les pointeurs vers les fichiers Delta sont mis à jour et les données en cache sont invalidées pour refléter l'état le plus récent.

V-Order : L'Optimisation Propriétaire pour Direct Lake

Pour maximiser les performances de Direct Lake, Microsoft a introduit V-Order, un algorithme propriétaire d'optimisation de l'écriture des fichiers Parquet. V-Order réorganise les données dans les fichiers Parquet de manière à les rendre plus rapidement scannables par le moteur VertiPaq, tout en maintenant la compatibilité avec le standard open source Parquet.

V-Order fonctionne de manière similaire à ce que VertiPaq fait pour obtenir une compression et des performances de requête optimales dans le mode Import : les données sont encodées en dictionnaire et compactées de façon similaire. Le résultat est que l'exécution des requêtes Direct Lake atteint des performances quasi équivalentes au mode Import.

Tous les moteurs de calcul Fabric (notebooks Spark, pipelines, Dataflow Gen2) créent automatiquement des tables Delta avec V-Order activé par défaut. Pour les données écrites depuis l'extérieur de Fabric, la configuration Spark suivante peut être utilisée :

```
# Activation de V-Order pour les écritures Spark
spark.conf.set("spark.sql.parquet.vorder.enabled", "true")

# Écriture d'un DataFrame avec V-Order
df.write \
  .format("delta") \
  .option("vorder.enabled", "true") \
  .mode("overwrite") \
  .save("/chemin/vers/table")
```

> **Performance**
> Les fichiers Parquet écrits sans V-Order fonctionnent toujours avec Direct Lake, mais les performances peuvent être significativement réduites. Pour les tables Iceberg virtualisées via raccourcis OneLake, V-Order n'est généralement pas appliqué aux fichiers source. Deux options s'offrent aux architectes : (1) accepter des performances légèrement réduites pour ces tables, ou (2) créer des copies matérialisées avec V-Order pour les tables critiques.

Direct Lake pour les Tables Iceberg Virtualisées

Une question naturelle se pose : le mode Direct Lake fonctionne-t-il avec les tables Iceberg accédées via raccourcis OneLake? La réponse est nuancée.

Direct Lake nécessite des tables Delta Lake dans OneLake. Les tables Iceberg virtualisées comme Delta via les raccourcis satisfont cette exigence — du point de vue de Power BI, elles apparaissent comme des tables Delta standard. Le modèle sémantique peut donc être créé en mode Direct Lake sur ces tables virtualisées.

Cependant, plusieurs considérations s'appliquent :

Latence de virtualisation. Chaque accès implique la conversion des métadonnées Iceberg vers Delta. Pour les tables fréquemment mises à jour, cette conversion ajoute une légère latence.

Absence de V-Order. Les fichiers Parquet écrits par des moteurs Iceberg externes n'utilisent généralement pas V-Order. Les performances de chargement en mémoire peuvent être inférieures à celles des tables Delta natives Fabric.

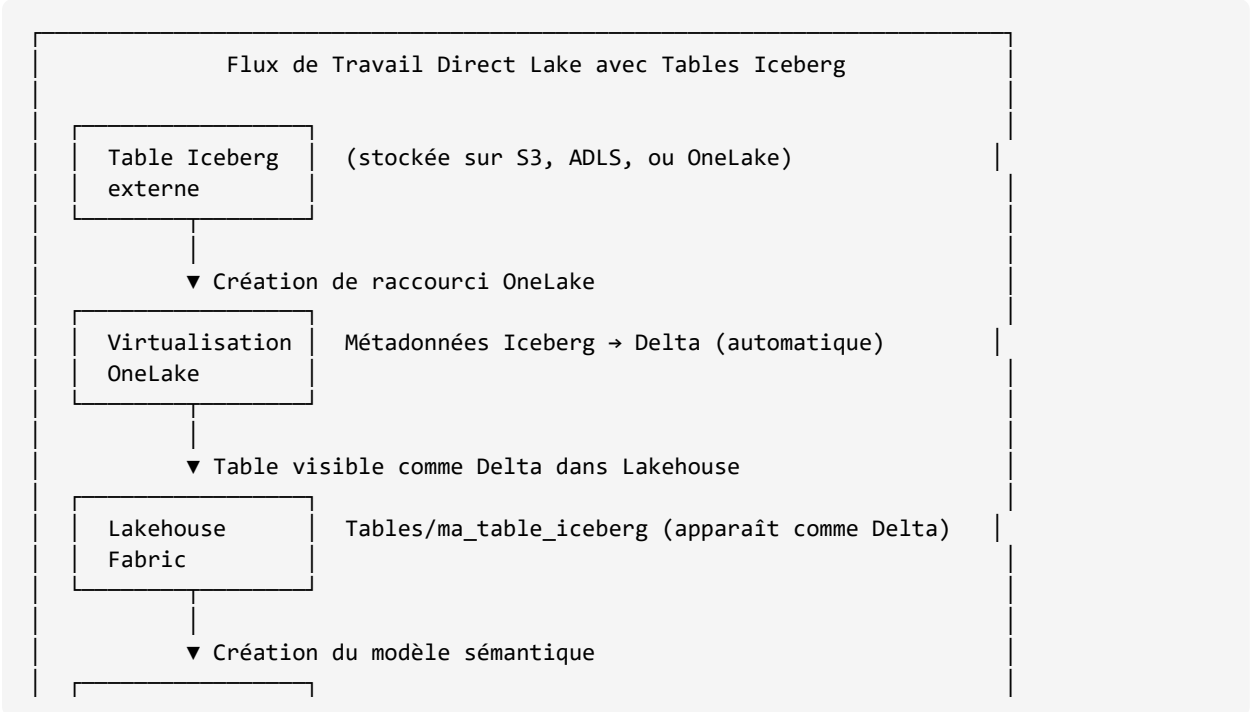
Optimisation des fichiers sources. Les performances Direct Lake dépendent fortement de la disposition des fichiers Parquet. Les bonnes pratiques incluent : tri des données par les colonnes de date fréquemment filtrées, taille de groupe de lignes (row group) optimisée, et partitionnement approprié.

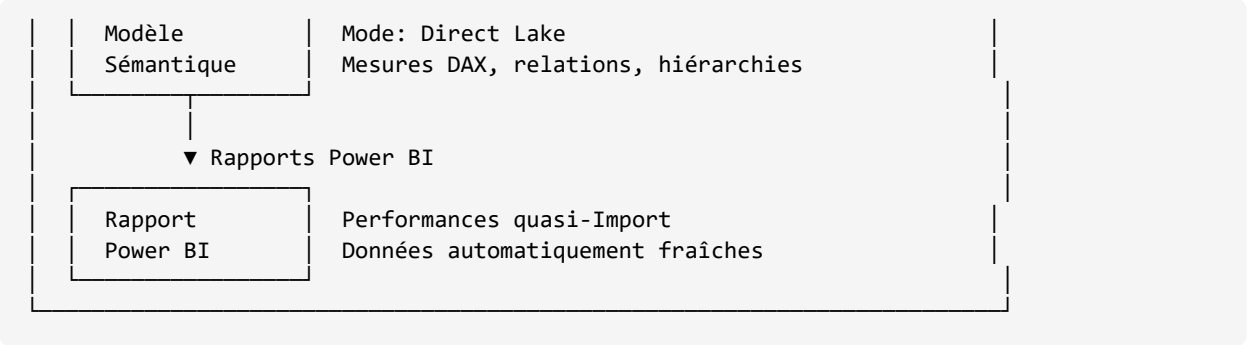
Pour les charges de travail Power BI critiques alimentées par des tables Iceberg, une approche hybride peut être considérée : utiliser les raccourcis pour les données moins fréquemment accédées, et matérialiser les tables les plus utilisées en Delta natif avec V-Order pour des performances optimales.

Configuration d'un Modèle Sémantique Direct Lake

La création d'un modèle sémantique Direct Lake s'effectue depuis l'interface Fabric. Voici le processus détaillé :

1. **Prérequis.** Disposer d'un Lakehouse avec des tables Delta (natives ou virtualisées depuis Iceberg) dans la section Tables.
2. **Création du modèle.** Depuis le Lakehouse, cliquer sur « Nouveau modèle sémantique ». Sélectionner les tables à inclure dans le modèle.
3. **Vérification du mode.** Dans les propriétés avancées du modèle, confirmer que le mode de stockage est bien « Direct Lake ».
4. **Modélisation.** Définir les relations entre tables, créer les mesures DAX, configurer les hiérarchies. Ces opérations s'effectuent via l'interface web de modélisation ou via des outils externes utilisant XMLA.
5. **Création de rapports.** Les rapports Power BI peuvent être créés directement depuis le modèle, bénéficiant immédiatement des performances Direct Lake.





Gestion du Repli DirectQuery

Direct Lake dispose d'un mécanisme de repli (fallback) vers DirectQuery lorsque certaines conditions empêchent le chargement direct des données. Ce repli peut survenir dans plusieurs situations :

Dépassement des limites de mémoire. Chaque SKU Fabric dispose de limites de mémoire pour les modèles Direct Lake. Si les données requises dépassent cette limite, le système bascule vers DirectQuery pour exécuter la requête.

Constructions DAX non supportées. Certaines expressions DAX complexes peuvent nécessiter un repli. Les colonnes calculées référençant des tables Direct Lake, par exemple, ne sont pas supportées.

Vues SQL non matérialisées. Direct Lake requiert des tables Delta physiques. Les vues définies au niveau du point de terminaison SQL déclenchent un repli.

Le comportement de repli est configurable via la propriété « Direct Lake behavior » du modèle sémantique :

- **Automatic** : Repli automatique vers DirectQuery si nécessaire. Certaines requêtes peuvent être lentes.
- **DirectLakeOnly** : Aucun repli. Les requêtes qui ne peuvent pas être satisfaites en Direct Lake retournent une erreur.
- **DirectQueryOnly** : Force toutes les requêtes à utiliser DirectQuery, désactivant effectivement Direct Lake.

Pour les modèles alimentés par des tables Iceberg virtualisées, la recommandation est d'utiliser le mode « DirectLakeOnly » pour les charges de travail de production critiques, et de s'assurer que toutes les mesures DAX sont compatibles avec les restrictions Direct Lake.

> **Migration**
> *De* : Modèle sémantique Power BI en mode Import avec actualisation nocturne
> *Vers* : Modèle Direct Lake sur tables Lakehouse
> *Stratégie* : Migration progressive. Commencer par recréer les tables sources dans le Lakehouse (via pipelines Data Factory ou Dataflow Gen2). Recréer le modèle sémantique en mode Direct Lake. Valider les performances et l'exactitude des mesures DAX. Migrer les rapports existants vers le nouveau modèle. Conserver temporairement l'ancien modèle en parallèle pour comparaison.

Power BI Embedded avec Direct Lake

Pour les éditeurs de logiciels indépendants (ISV) et les développeurs intégrant des analyses Power BI dans leurs applications, le mode Direct Lake est désormais disponible pour Power BI Embedded. Annoncé en

prévisualisation début 2025 et en disponibilité générale en mars 2025, cette fonctionnalité permet aux applications embarquées de bénéficier des mêmes avantages de performance et de fraîcheur des données.

Les avantages clés pour les scénarios embarqués incluent :

Performance améliorée. Le chargement direct des fichiers Parquet en mémoire offre l'expérience de requête et de rapport la plus rapide.

Insights en temps réel. Les mises à jour des données sources sont automatiquement reflétées sans nécessiter d'actualisations périodiques.

Réduction des coûts opérationnels. L'élimination des actualisations planifiées réduit la consommation de ressources de capacité et les risques d'échec d'actualisation.

Considérations de Coûts et de Capacité

L'adoption de Direct Lake avec des tables Iceberg virtualisées implique des considérations de coûts spécifiques que les architectes doivent évaluer.

Capacité Fabric. Direct Lake requiert une capacité Fabric (F2 et supérieur, ou Power BI Premium P1 et supérieur). Les limites de mémoire et de performance varient selon le SKU :

SKU	Mémoire Max par Modèle	Lignes Max par Table	Parallélisme Max
F2	3 Go	300 millions	4
F4	3 Go	300 millions	8
F8	3 Go	300 millions	16
F16	6 Go	600 millions	32
F32	12 Go	1.2 milliard	64
F64	24 Go	2.4 milliards	128
F128	48 Go	4.8 milliards	256
F256	96 Go	9.6 milliards	512
F512	192 Go	19.2 milliards	1024

Ces limites sont importantes pour le dimensionnement. Pour les tables Iceberg virtualisées contenant des milliards de lignes, un SKU F64 ou supérieur peut être nécessaire pour éviter les replis vers DirectQuery.

Facteurs influençant le choix du SKU :

Volume de données actif. Le modèle sémantique ne charge que les colonnes interrogées, mais les tables de dimension complètes sont souvent chargées. Estimer le volume des tables de dimension plus les colonnes de faits fréquemment utilisées.

Complexité des mesures DAX. Les mesures impliquant des calculs sur de grandes tables nécessitent plus de mémoire temporaire. Les mesures utilisant CALCULATE avec de nombreux filtres ou SUMMARIZE sur des millions de lignes augmentent les besoins.

Concurrence des utilisateurs. Chaque session utilisateur active consomme de la mémoire pour son contexte de requête. Les tableaux de bord avec de nombreux utilisateurs simultanés nécessitent plus de mémoire disponible.

Actualisation de page automatique. Si configurée, l'actualisation automatique des visuels génère des requêtes régulières qui maintiennent les données en mémoire et consomment de la capacité.

Coûts de stockage OneLake. Le stockage dans OneLake est facturé séparément de la capacité de calcul. Pour les données accédées via raccourcis depuis des stockages externes, les coûts de sortie (egress) du fournisseur source s'ajoutent.

Optimisation des coûts. Plusieurs stratégies permettent d'optimiser les coûts :

1. **Mise en pause automatique.** Les capacités Fabric peuvent être configurées pour se mettre en pause automatiquement lors des périodes d'inactivité.
2. **Dimensionnement approprié.** Choisir le SKU minimal satisfaisant les exigences de performance et de limites de données.
3. **Partitionnement intelligent.** Partitionner les tables par date permet aux requêtes de scanner uniquement les partitions pertinentes, réduisant le volume de données chargées.
4. **Matérialisation sélective.** Pour les tables Iceberg fréquemment accédées via raccourcis, évaluer si la matérialisation en Delta natif (avec V-Order) réduirait les coûts de sortie tout en améliorant les performances.

> **Étude de cas : Institution financière de Toronto**
> *Secteur* : Services financiers
> *Défi* : Réduire la latence des tableaux de bord de risque alimentés par un entrepôt de données Snowflake contenant 50 To de données historiques.
> *Solution* : Configuration de Fabric avec capacité F64 dans la région Canada Central. Les tables de faits principales (transactions, positions) restent dans Snowflake et sont accessibles via raccourcis OneLake comme tables Iceberg virtualisées. Les tables de dimensions sont matérialisées dans le Lakehouse Fabric avec V-Order. Modèle sémantique Direct Lake avec agrégations précalculées pour les métriques les plus demandées.
> *Résultats* : Temps de réponse des tableaux de bord réduit de 15 secondes à moins de 2 secondes. Coût mensuel de la solution Fabric compensé par la réduction des actualisations nocturnes et l'élimination de deux serveurs Analysis Services dédiés.

Le Partenariat Microsoft-Snowflake : Interopérabilité Native

Le partenariat entre Microsoft et Snowflake, annoncé en 2024 et progressivement mis en œuvre depuis, représente une avancée majeure pour l'interopérabilité des lakehouses. Ce partenariat permet aux utilisateurs de Snowflake et Microsoft Fabric de travailler sur les mêmes données Iceberg dans OneLake, sans duplication ni mouvement de données.

Fonctionnalités en disponibilité générale (2025) :

- Traduction automatique des métadonnées Iceberg vers Delta Lake pour utilisation avec tous les moteurs Fabric
- Raccourcis vers les données Iceberg Snowflake stockées dans Azure, Amazon S3 ou Google Cloud Storage
- Stockage natif des données Iceberg Snowflake directement dans OneLake
- Support Iceberg dans la fonctionnalité de miroir (mirroring) Snowflake

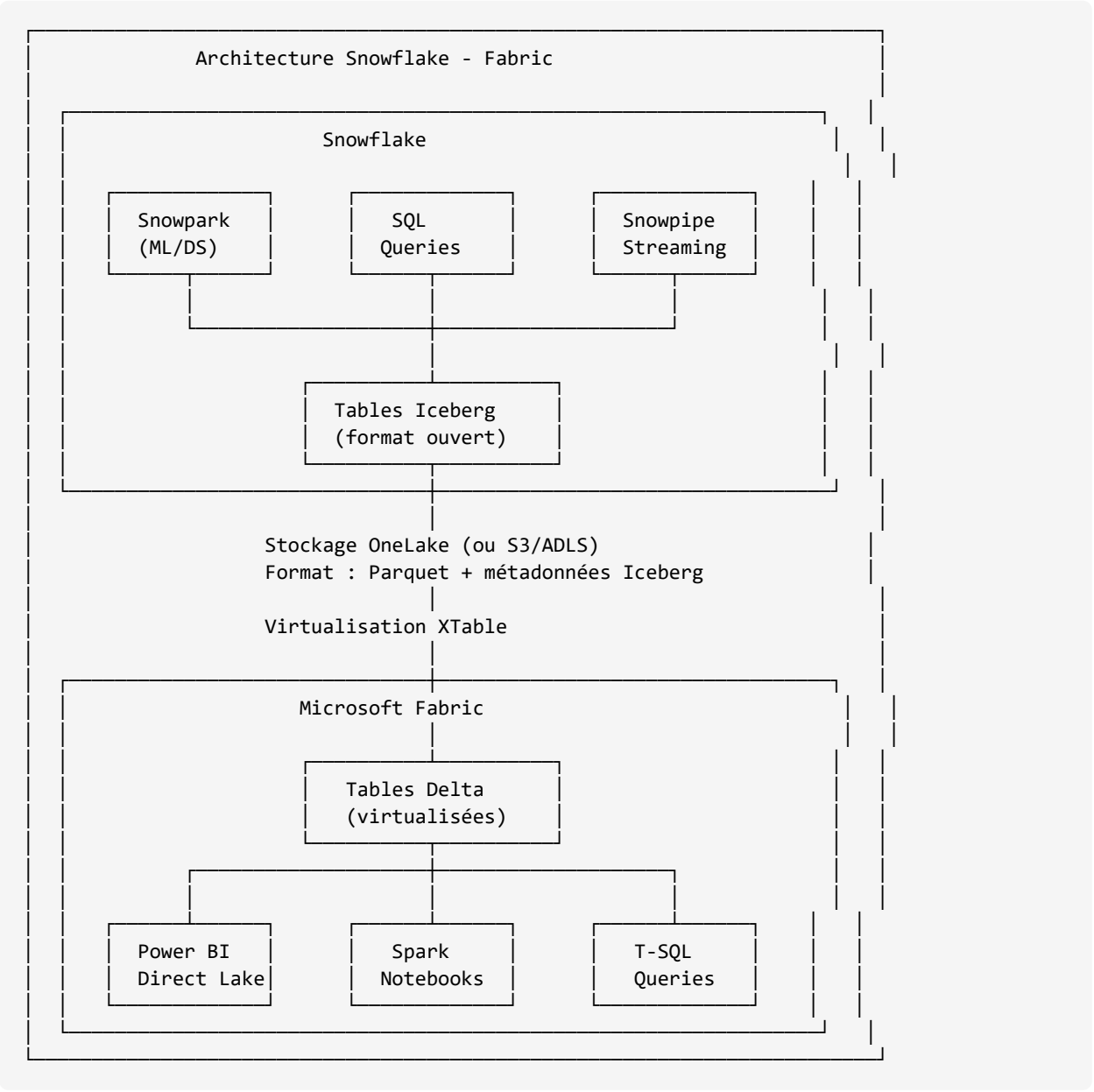
Fonctionnalités en prévisualisation :

- Conversion automatique des données Fabric vers le format Iceberg pour utilisation dans Snowflake

- Nouvelles APIs de table OneLake intégrées avec la fonctionnalité de base de données liée au catalogue de Snowflake

Cette intégration répond à un besoin réel des organisations : pouvoir utiliser le meilleur outil pour chaque tâche sans être contraint par les silos de données. Les équipes data science peuvent utiliser Snowpark pour l'entraînement de modèles ML, tandis que les analystes métier créent des rapports dans Power BI — le tout sur les mêmes données sans duplication.

Architecture du partenariat :



La Fonctionnalité de Miroir (Mirroring) Snowflake

Le miroir Snowflake dans Fabric permet de répliquer automatiquement les données de Snowflake vers OneLake au format Delta, sans nécessiter de pipelines ETL. Cette fonctionnalité, en disponibilité générale avec support Iceberg depuis novembre 2025, offre une alternative aux raccourcis pour les scénarios nécessitant des performances optimales.

Différences entre raccourcis et miroir :

Aspect	Raccourcis	Miroir
Duplication des données	Non	Oui (copie dans OneLake)
Latence des données	Temps réel	Quasi temps réel (minutes)
Coûts de sortie	Oui (si source externe)	Une seule fois lors de la copie
Format dans OneLake	Virtualisé (via XTable)	Delta natif avec V-Order
Performance Direct Lake	Bonne	Optimale
Configuration	Simple	Nécessite configuration du miroir

Quand utiliser le miroir plutôt que les raccourcis :

- Les performances Power BI sont critiques et doivent être optimales
- Les coûts de sortie récurrents dépassent le coût de stockage OneLake
- Les tables sont stables et ne nécessitent pas une fraîcheur au niveau de la seconde
- L'organisation souhaite bénéficier de la gouvernance OneLake complète

Limitations et Contraintes Actuelles

Malgré les avancées significatives, l'intégration Fabric-Iceberg présente certaines limitations que les architectes doivent connaître lors de la conception de leurs solutions.

Limitations des raccourcis Iceberg :

Types de données. Tous les types de données Iceberg ne sont pas encore supportés pour la conversion vers Delta. Les types complexes comme les structures imbriquées profondes ou certains types temporels peuvent poser des problèmes. Il est recommandé de valider la compatibilité des schémas avant la mise en production.

Fonctionnalités Iceberg avancées. Certaines fonctionnalités avancées d'Iceberg comme les branches et les étiquettes (tags) ne sont pas traduites vers Delta. Les tables utilisant ces fonctionnalités peuvent être accessibles, mais ces métadonnées spécifiques ne seront pas disponibles dans Fabric.

Vecteurs de suppression. Bien que Microsoft ait amélioré la conversion des vecteurs de suppression Delta vers les fichiers de suppression positionnelle Iceberg, certains cas limites peuvent entraîner des incohérences temporaires lors de la conversion.

Limitations Direct Lake :

Colonnes calculées. Les colonnes calculées DAX référençant des tables Direct Lake ne sont pas supportées. Cette logique doit être implémentée en amont, dans la couche de transformation du Lakehouse.

Tables calculées. De même, les tables calculées en DAX ne sont pas supportées. Utiliser des vues matérialisées ou des tables dérivées dans le Lakehouse.

Certaines fonctions DAX. Quelques fonctions DAX avancées peuvent déclencher un repli vers DirectQuery. Tester les mesures complexes pour vérifier leur compatibilité.

Vues SQL non matérialisées. Direct Lake requiert des tables Delta physiques. Les vues définies uniquement au niveau du point de terminaison SQL déclenchent un repli.

Sécurité au niveau des lignes via SQL. La sécurité RLS définie sur le point de terminaison SQL n'est pas appliquée en mode Direct Lake. Utiliser plutôt la sécurité OneLake ou la RLS au niveau du modèle sémantique.

Tableau récapitulatif des limitations :

Fonctionnalité	Raccourcis Iceberg	Direct Lake
Types complexes imbriqués	Partiel	N/A
Branches/Tags Iceberg	Non	N/A
Time Travel Iceberg	Non	N/A
Colonnes calculées DAX	N/A	Non
Tables calculées DAX	N/A	Non
Vues SQL	N/A	Repli DirectQuery
RLS via SQL endpoint	N/A	Non appliquée
Actualisation incrémentielle	N/A	Automatique

> **Performance**
> Pour contourner les limitations des colonnes calculées, créer des colonnes dérivées directement dans les tables Delta/Iceberg lors de l'ingestion. Cette approche offre de meilleures performances car les calculs sont effectués une seule fois lors de l'écriture plutôt qu'à chaque requête.

Dépannage et Résolution des Problèmes Courants

L'intégration Fabric-Iceberg peut présenter des défis techniques. Voici les problèmes les plus fréquents et leurs solutions.

Problème : Échec de création du raccourci vers une table Iceberg

Symptômes : Message d'erreur lors de la création du raccourci, table non visible dans le Lakehouse.

Causes possibles et solutions : 1. **Permissions insuffisantes** : Vérifier que les credentials configurées ont accès en lecture au bucket/container et à tous les fichiers de métadonnées Iceberg. 2. **Structure de table non standard** : S'assurer que le dossier sélectionné est bien le dossier racine de la table Iceberg, contenant les sous-dossiers `metadata/` et `data/`. 3. **Version de protocole non supportée** : Vérifier que la table utilise une version du protocole Iceberg supportée par OneLake.

Problème : Performances Direct Lake dégradées sur tables virtualisées

Symptômes : Temps de réponse lents, chargement initial prolongé des rapports.

Causes possibles et solutions : 1. **Absence de V-Order** : Les fichiers Parquet Iceberg ne bénéficient pas de V-Order. Considérer la matérialisation en Delta natif pour les tables critiques. 2. **Fichiers fragmentés** : Trop de petits fichiers Parquet ralentissent le scan. Exécuter une compaction sur les tables Iceberg sources. 3. **Partitionnement inefficace** : Vérifier que le partitionnement des tables Iceberg correspond aux patterns de filtrage des rapports.

Problème : Repli DirectQuery fréquent

Symptômes : Les requêtes utilisent DirectQuery au lieu de Direct Lake, performances inconsistantes.

Causes possibles et solutions : 1. **Dépassement des limites** : Vérifier que le volume de données est compatible avec les limites du SKU Fabric. 2. **Mesures DAX incompatibles** : Identifier les mesures déclenchant le repli via les traces de requête et les réécrire. 3. **Comportement Direct Lake** : Vérifier la configuration « DirectLakeOnly » vs « Automatic » dans les paramètres du modèle.

Problème : Données obsolètes dans les rapports Direct Lake

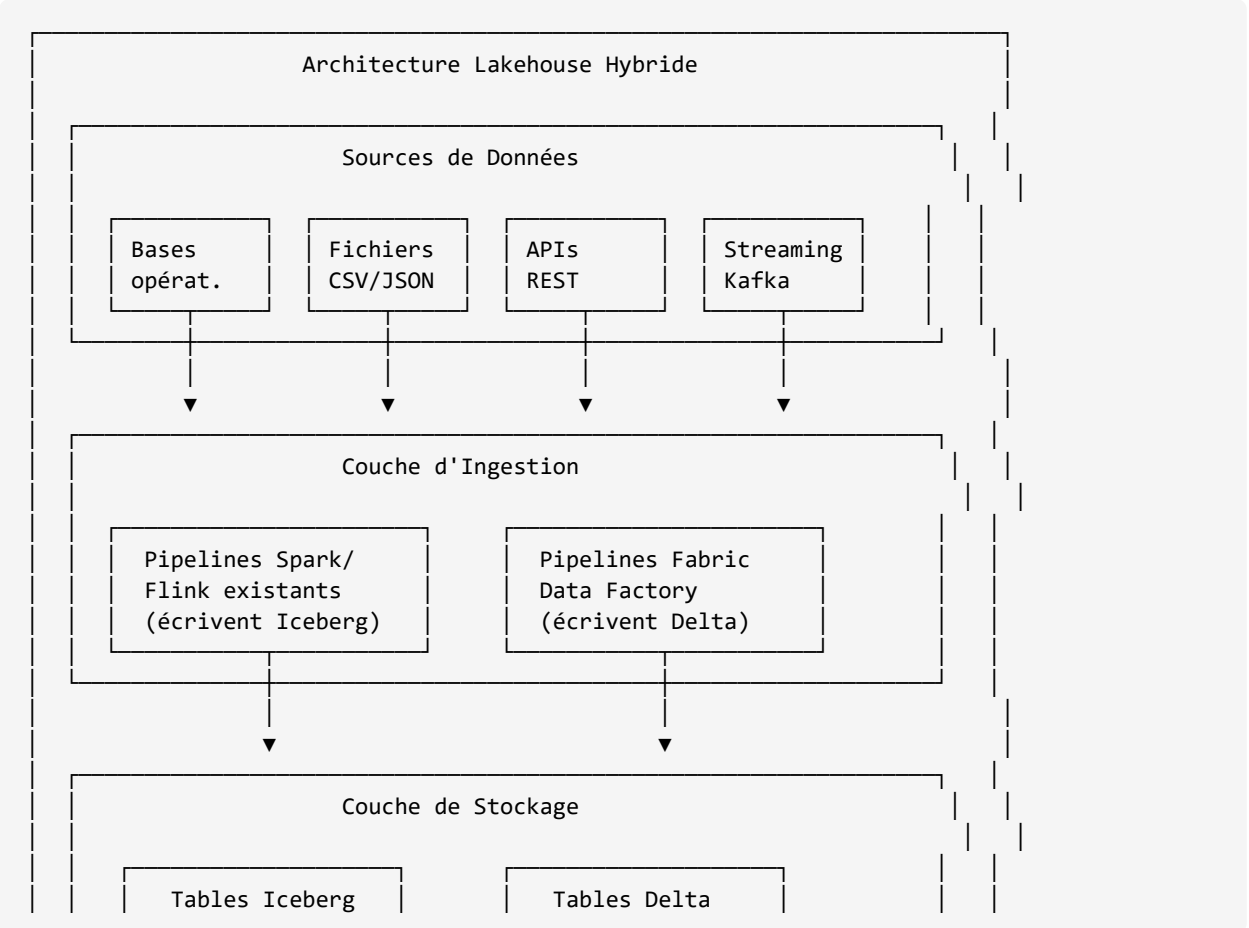
Symptômes : Les rapports ne reflètent pas les dernières modifications des tables Iceberg.

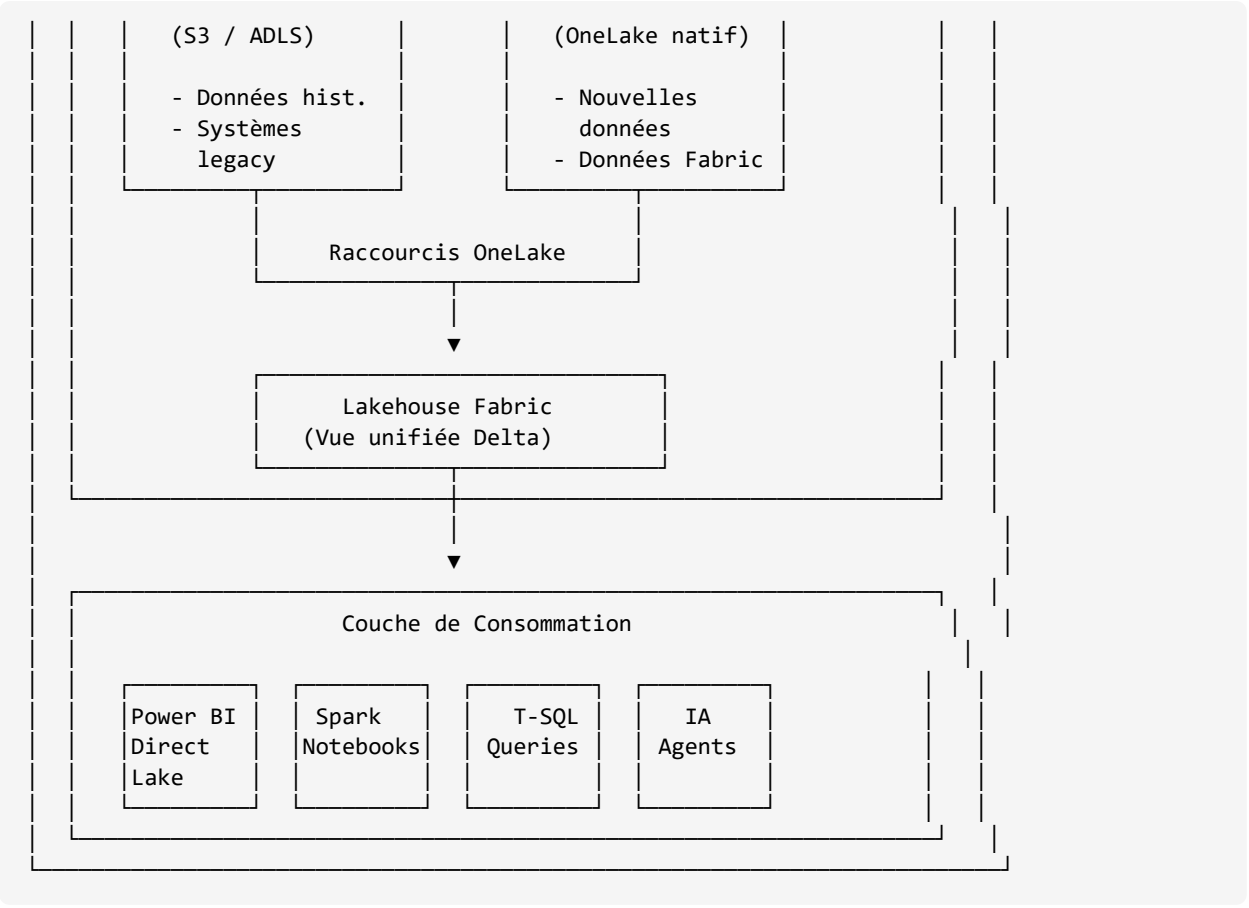
Causes possibles et solutions : 1. **Cache de métadonnées** : La virtualisation Iceberg → Delta peut avoir un délai. Attendre quelques minutes après les mises à jour de la table source. 2. **Opération de cadrage (framing)** : Exécuter manuellement une actualisation du modèle sémantique pour forcer la détection des nouveaux fichiers. 3. **Détection de changements** : Vérifier dans les journaux de conversion (`latest_conversion_log.txt`) que la conversion s'effectue correctement.

IV.14.3 Patterns d'Architecture pour l'Intégration Fabric-Iceberg

Pattern 1 : Lakehouse Hybride Multi-Format

Ce pattern convient aux organisations disposant d'investissements existants significatifs dans Apache Iceberg qu'elles souhaitent intégrer progressivement à Microsoft Fabric.



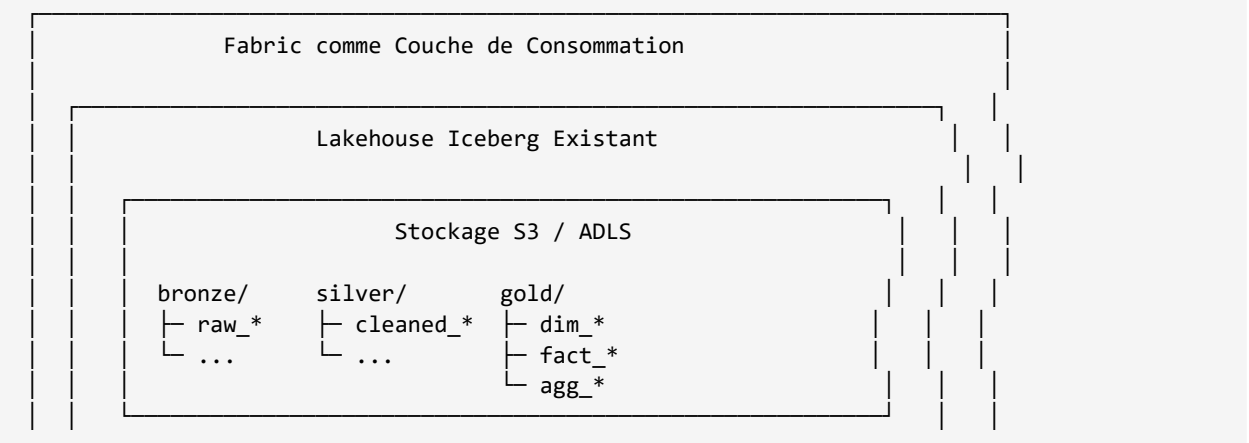


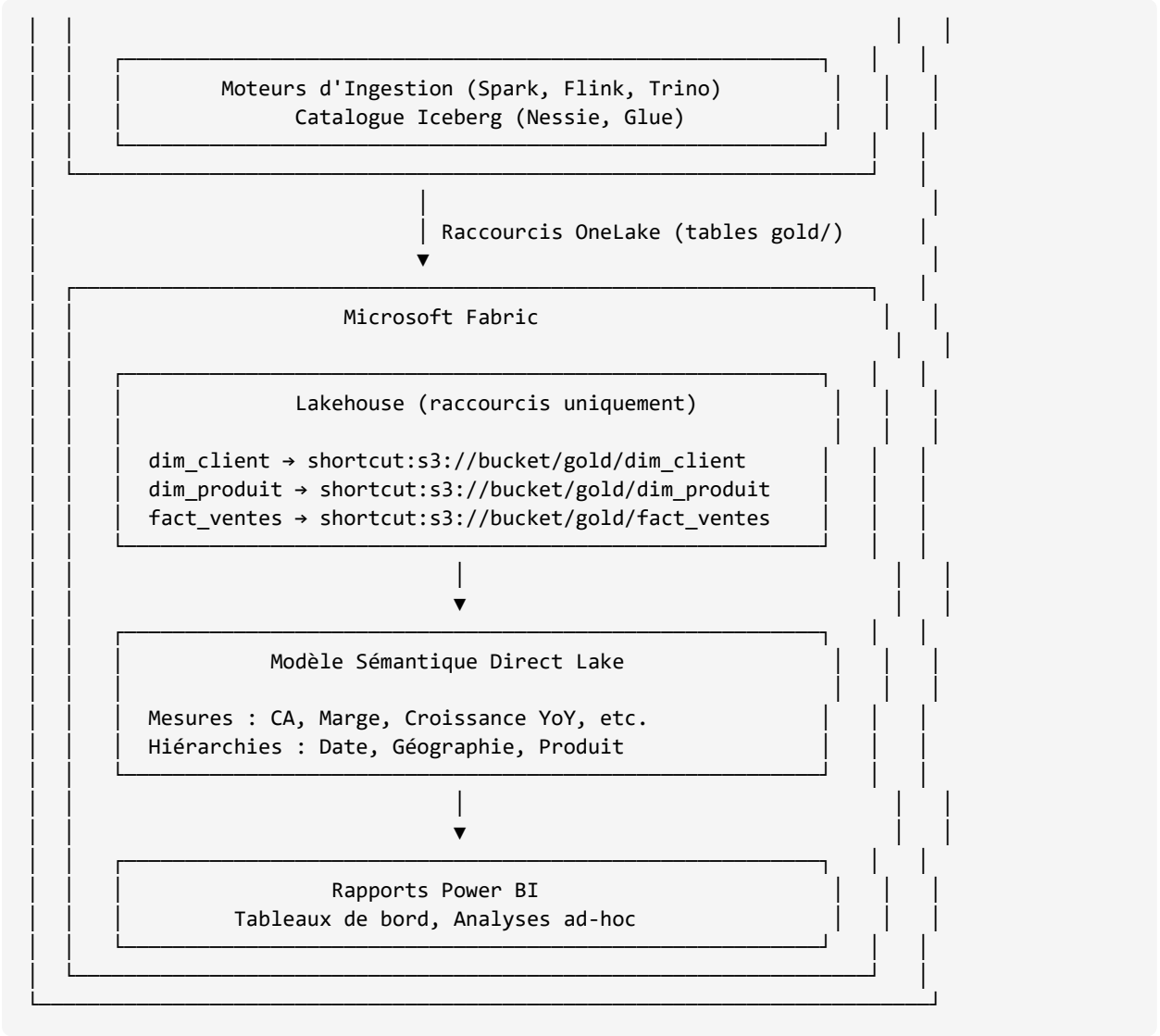
Avantages : - Préservation des investissements existants dans les pipelines Iceberg - Migration progressive sans perturbation des systèmes en production - Flexibilité pour choisir le meilleur moteur d'écriture selon le cas d'usage

Considérations : - Complexité de gouvernance avec deux formats de table - Nécessité de maintenir l'expertise sur les deux écosystèmes - Potentielle incohérence dans l'optimisation des fichiers (V-Order absent pour Iceberg)

Pattern 2 : Fabric comme Couche de Consommation Unifiée

Ce pattern positionne Fabric exclusivement comme couche de consommation, laissant l'ingestion et le stockage primaire à l'écosystème Iceberg existant.



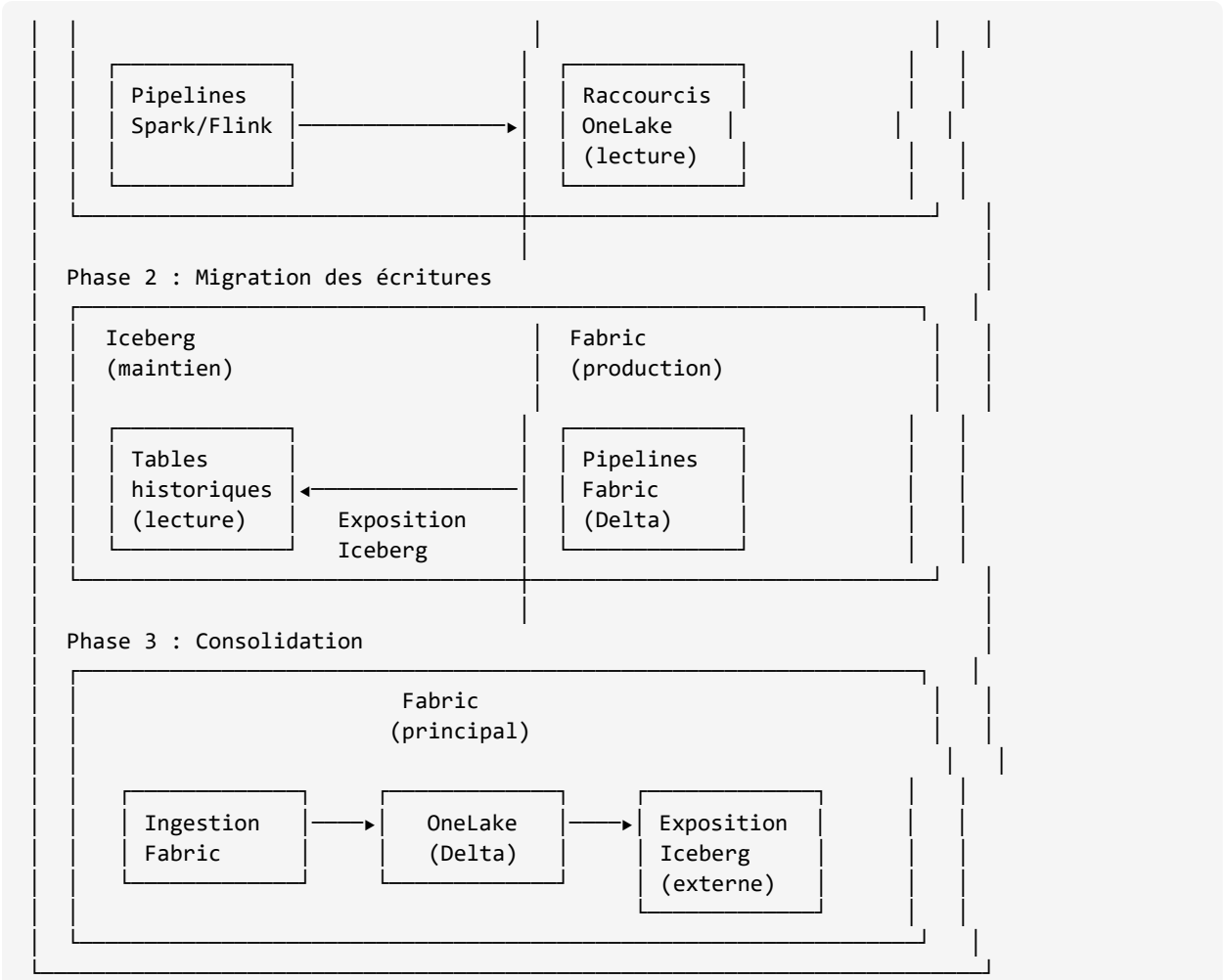


Avantages : - Séparation claire des responsabilités : Iceberg pour l'ingestion, Fabric pour la consommation - Aucune modification des pipelines d'ingestion existants - Coûts Fabric limités à la capacité de calcul (pas de stockage primaire)

Considérations : - Dépendance aux performances réseau pour l'accès aux données - Coûts de sortie (egress) si les données sont sur un fournisseur différent d'Azure - Performances Direct Lake potentiellement réduites sans V-Order

Pattern 3 : Migration Progressive vers Fabric Natif

Ce pattern convient aux organisations souhaitant migrer progressivement leur Lakehouse Iceberg vers Fabric, tout en maintenant la compatibilité avec les systèmes existants.



Phase 1 - Coexistence (3-6 mois) : - Création des raccourcis OneLake vers les tables Iceberg existantes - Développement des modèles sémantiques Direct Lake - Validation des performances et de l'exactitude

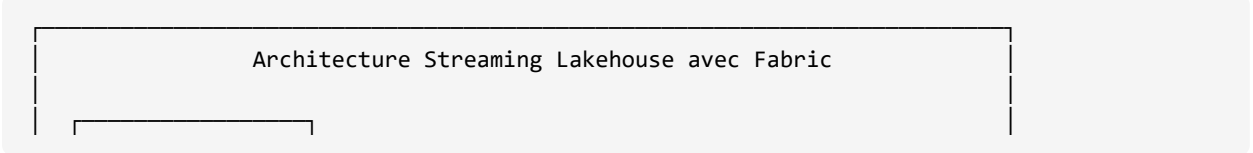
Phase 2 - Migration des écritures (6-12 mois) : - Nouveaux pipelines développés nativement dans Fabric - Tables existantes migrées progressivement vers Delta avec V-Order - Exposition des nouvelles tables en Iceberg pour les consommateurs externes

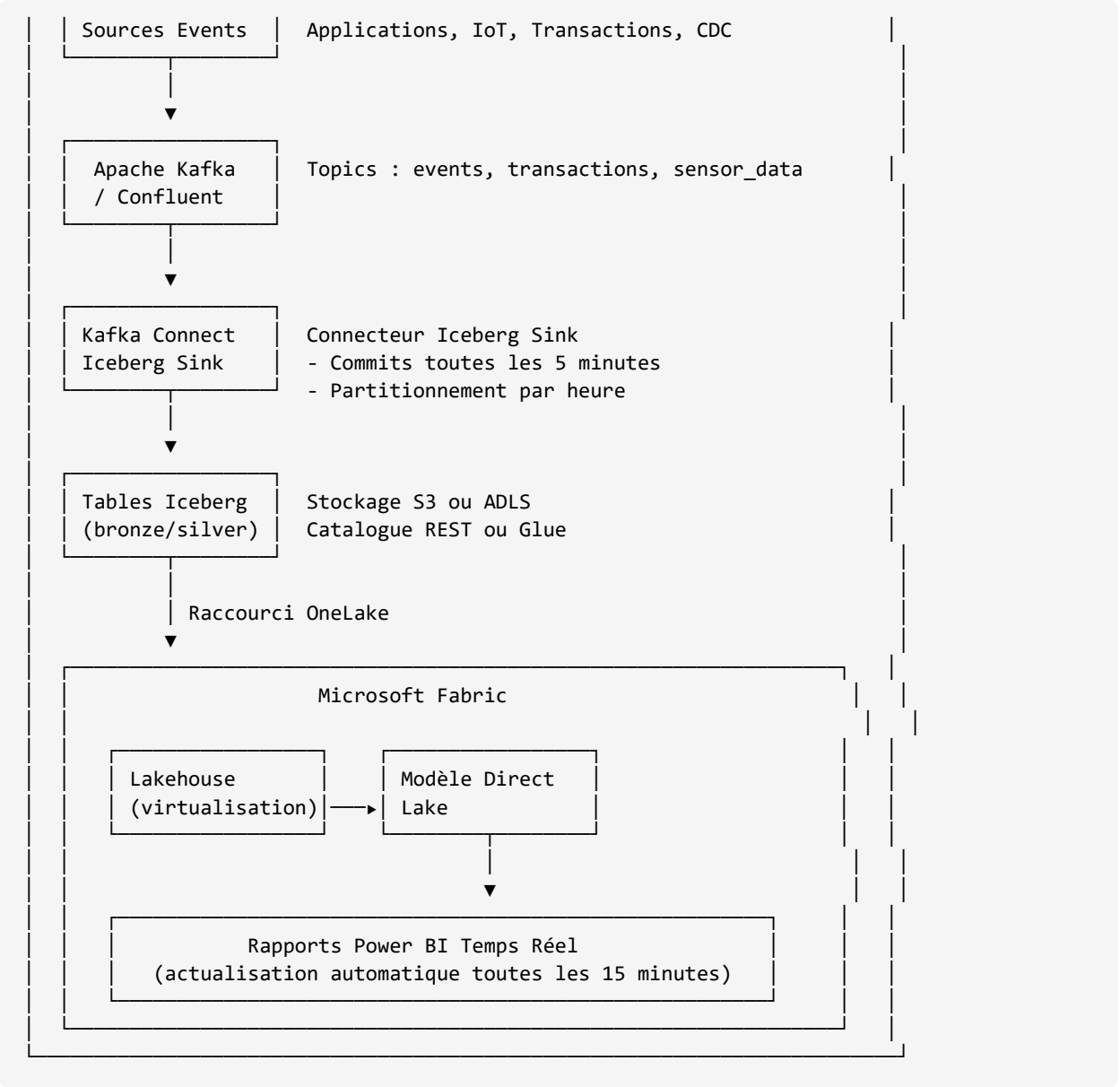
Phase 3 - Consolidation (12-18 mois) : - Fabric devient la plateforme principale - Décommissionnement des anciens pipelines Iceberg - Maintien de l'exposition Iceberg pour l'interopérabilité

Intégration avec le Streaming Lakehouse

L'intégration de Microsoft Fabric avec Apache Iceberg s'inscrit naturellement dans l'architecture de Streaming Lakehouse décrite au Volume III de cette monographie. Les tables Iceberg alimentées par des pipelines Apache Kafka peuvent être exposées dans Fabric pour l'analytique en temps quasi réel.

Un pattern courant combine Confluent Cloud ou Apache Kafka pour l'ingestion streaming, des connecteurs Kafka vers Iceberg pour la persistance, et Microsoft Fabric pour la consommation analytique :





Intégration avec les Agents de Données IA et Copilot

L'une des propositions de valeur les plus innovantes de Microsoft Fabric est l'intégration native avec les capacités d'intelligence artificielle, incluant Copilot et les agents de données. Cette intégration s'étend naturellement aux tables Iceberg virtualisées, ouvrant de nouvelles possibilités pour l'analyse conversationnelle des données.

Fabric Data Agents. Les agents de données Fabric permettent aux utilisateurs d'interroger leurs données en langage naturel. Ces agents peuvent raisonner sur les données stockées dans les Lakehouses, incluant les tables Iceberg accessibles via raccourcis. L'intégration avec Microsoft 365 Copilot permet d'accéder à ces données directement depuis les applications de productivité.

Fonctionnement avec les tables Iceberg virtualisées :

1. **Découverte des données** : L'agent de données découvre les tables disponibles dans le Lakehouse, incluant les raccourcis vers les tables Iceberg.
2. **Compréhension du schéma** : L'agent analyse les schémas des tables pour comprendre la structure des données.

3. **Génération de requêtes** : En réponse aux questions en langage naturel, l'agent génère des requêtes SQL ou DAX appropriées.
4. **Exécution et présentation** : Les requêtes sont exécutées contre les tables (via la couche de virtualisation pour Iceberg) et les résultats sont présentés à l'utilisateur.

Préparation des données pour l'IA (« Prep for AI ») :

Pour optimiser l'expérience des agents de données, Fabric offre la fonctionnalité « Prep for AI » qui permet de personnaliser la façon dont les modèles sémantiques sont présentés aux modèles d'IA :

- **Descriptions enrichies** : Ajouter des descriptions métier aux tables et colonnes pour améliorer la compréhension de l'agent.
- **Synonymes** : Définir des termes alternatifs que les utilisateurs pourraient employer.
- **Exemples de questions** : Fournir des exemples de questions typiques pour guider l'agent.
- **Règles métier** : Documenter les règles de calcul spécifiques au domaine.

Ces personnalisations sont automatiquement respectées lorsque l'agent de données accède au modèle sémantique, y compris pour les tables provenant de raccourcis Iceberg.

> **Étude de cas : Compagnie d'assurance de Vancouver**
> *Secteur* : Services financiers - Assurance
> *Défi* : Permettre aux actuaires d'analyser les données de sinistres via des questions en langage naturel, sans nécessiter de compétences SQL avancées.
> *Solution* : Les tables de sinistres historiques, stockées au format Iceberg dans un Data Lake AWS S3 existant, sont exposées dans Fabric via raccourcis. Un modèle sémantique Direct Lake est créé avec des mesures actuarielles (ratio de sinistralité, provisions, etc.). La fonctionnalité « Prep for AI » est utilisée pour ajouter le vocabulaire actuariel. Un agent de données permet aux actuaires de poser des questions comme « Quel est le ratio de sinistralité pour l'automobile en Ontario en 2024? »
> *Résultats* : Réduction de 70% du temps nécessaire pour obtenir des analyses ad-hoc. Les actuaires peuvent explorer les données sans attendre les rapports des équipes BI. La conformité avec les exigences du BSIF est maintenue grâce à la sécurité OneLake.

Cette architecture permet d'exploiter les données Iceberg existantes

> **Étude de cas : Détaillant québécois**
> *Secteur* : Commerce de détail
> *Défi* : Offrir des tableaux de bord en temps réel des ventes et de l'inventaire pour les 200 magasins, tout en maintenant les systèmes analytiques existants basés sur Trino et Iceberg.
> *Solution* : Architecture streaming avec Kafka ingérant les événements de point de vente. Connecteur Iceberg Sink écrivant les tables dans ADLS. Raccourcis OneLake exposant les tables dans Fabric. Modèle sémantique Direct Lake avec actualisation automatique. Les analystes Trino existants continuent d'interroger les mêmes tables Iceberg.
> *Résultats* : Latence des tableaux de bord réduite de 24 heures à 15 minutes. Maintien de la compatibilité avec les rapports Trino existants. Réduction de 60% du temps de développement des nouveaux rapports grâce à l'interface Power BI en libre-service.

IV.14.4 Recommandations et Bonnes Pratiques

Choix de l'Architecture d'Intégration

Le choix entre les différents patterns d'intégration dépend de plusieurs facteurs organisationnels et techniques.

Favoriser les raccourcis sans migration lorsque : - Les pipelines Iceberg existants sont stables et performants - Les coûts de sortie (egress) sont acceptables - L'équipe ne dispose pas des ressources pour une migration complète - La flexibilité multi-moteurs (Trino, Dremio, Snowflake) est requise

Favoriser la migration vers Delta natif lorsque : - Les performances Power BI sont critiques (bénéfice de V-Order) - Les données sont principalement consommées via Fabric - Les coûts de sortie actuels sont élevés - L'organisation souhaite consolider sur une plateforme unique

Favoriser l'architecture hybride lorsque : - L'organisation est en transition entre deux plateformes - Différentes équipes ont des préférences d'outils différentes - Une migration progressive est préférée à un basculement

Optimisation des Performances

Pour maximiser les performances de l'intégration Fabric-Iceberg :

1. **Optimisation des fichiers Parquet sources.** Pour les tables Iceberg consultées fréquemment, optimiser la disposition des fichiers : compaction régulière, tri par les colonnes de filtrage courantes, taille de fichiers entre 128 Mo et 1 Go.
2. **Partitionnement stratégique.** Utiliser le partitionnement masqué Iceberg par date pour les tables de faits. Cela permet aux requêtes Power BI de ne scanner que les partitions pertinentes.
3. **Gestion des agrégations.** Pour les requêtes très fréquentes, créer des tables d'agrégation précalculées. Ces tables peuvent être stockées directement en Delta avec V-Order pour des performances optimales.
4. **Surveillance des températures de colonnes.** Utiliser les vues de gestion dynamique (DMV) de Power BI pour identifier les colonnes fréquemment chargées. Optimiser le modèle pour minimiser le chargement de colonnes inutiles.

Gouvernance et Sécurité

L'intégration de données externes via raccourcis nécessite une attention particulière à la gouvernance :

1. **Inventaire des raccourcis.** Maintenir un inventaire documenté de tous les raccourcis, leurs sources, et leurs propriétaires. Utiliser les capacités du catalogue OneLake pour la découverte.
2. **Contrôles d'accès cohérents.** Définir les rôles de sécurité OneLake pour appliquer des restrictions uniformes, même sur les données externes.
3. **Lignage des données.** Documenter le flux de données depuis les sources Iceberg jusqu'aux rapports Power BI. Les outils de lignage Fabric facilitent cette traçabilité.
4. **Conformité réglementaire.** Pour les organisations canadiennes, s'assurer que les données sensibles restent dans les régions Azure canadiennes, même si elles sont accédées via raccourcis.

Surveillance et Opérations

1. **Métriques de performance Direct Lake.** Surveiller les temps de chargement des colonnes, les replis vers DirectQuery, et l'utilisation de la mémoire du modèle sémantique. Les vues de gestion dynamique (DMV) Analysis Services fournissent des informations détaillées sur les températures des colonnes et les statistiques de chargement.
2. **Latence de virtualisation.** Surveiller le temps de conversion des métadonnées Iceberg vers Delta. Des latences élevées peuvent indiquer des problèmes avec les tables sources. Le fichier `latest_conversion_log.txt` dans le dossier `_delta_log/` fournit des informations sur la dernière conversion.
3. **Diagnostics OneLake.** Activer les journaux de diagnostic OneLake pour répondre aux questions « qui a accédé à quoi, quand et comment ». Ces diagnostics, en disponibilité générale depuis 2025, diffusent des événements JSON vers un Lakehouse de votre choix, permettant une analyse approfondie avec Spark, SQL ou Power BI.
4. **Alertes d'actualisation.** Configurer des alertes pour détecter les échecs de synchronisation des métadonnées ou les problèmes de connectivité avec les stockages externes.
5. **Immutabilité des journaux.** Pour les organisations soumises à des exigences de conformité strictes, activer l'immutabilité des journaux de diagnostic OneLake. Cette fonctionnalité garantit que les événements ne peuvent être modifiés ou supprimés pendant une période de rétention définie.

Tableau de bord de surveillance recommandé :

Métrique	Source	Seuil d'alerte
Temps de chargement moyen Direct Lake	DMV Analysis Services	> 5 secondes
Taux de repli DirectQuery	Journaux d'audit Fabric	> 5% des requêtes
Utilisation mémoire modèle	Métriques de capacité	> 80% de la limite SKU
Latence conversion Iceberg→Delta	Fichier conversion_log	> 30 secondes
Erreurs de raccourcis	Diagnostics OneLake	Toute erreur
Temps de réponse P95 des rapports	Journaux Power BI	> 10 secondes

Considérations de Continuité d'Activité

L'intégration Fabric-Iceberg introduit des dépendances qui doivent être prises en compte dans la planification de la continuité d'activité.

Points de défaillance potentiels :

1. **Disponibilité du stockage source** : Si les tables Iceberg sont stockées dans un système externe (S3, GCS), la disponibilité de ce système impacte directement les rapports Fabric.
2. **Connectivité réseau** : Les raccourcis vers des sources externes dépendent de la connectivité réseau. Les interruptions réseau rendront les tables inaccessibles.

3. **Quotas et limites** : Les fournisseurs de stockage externe peuvent imposer des limites de débit qui, si atteintes, dégradent les performances ou causent des échecs.
4. **Authentification** : L'expiration des credentials ou des jetons d'authentification peut interrompre l'accès aux données.

Stratégies de résilience :

Pour les données critiques : Considérer la réplication dans OneLake via le miroir Snowflake ou des pipelines Fabric. Cela élimine la dépendance externe et garantit la disponibilité même si le système source est indisponible.

Pour les données moins critiques : Implémenter une surveillance proactive avec des alertes sur les échecs d'accès. Documenter les procédures de récupération et les contacts des équipes responsables des systèmes sources.

Redondance géographique : Pour les organisations nécessitant une haute disponibilité, considérer la réplication des données Iceberg dans plusieurs régions avec des raccourcis configurés vers la source la plus proche.

Évolutions Futures et Feuille de Route

L'intégration entre Microsoft Fabric et Apache Iceberg continue d'évoluer rapidement. Selon les annonces de Microsoft lors des événements Ignite 2025 et FabCon Vienna 2025, plusieurs améliorations sont prévues :

Court terme (2026) :

- Support étendu des types de données Iceberg complexes
- Amélioration des performances de virtualisation avec mise en cache des métadonnées
- Intégration avec les APIs REST Catalog Iceberg pour une interopérabilité améliorée
- Extension du support des raccourcis vers de nouvelles sources

Moyen terme (2026-2027) :

- Écriture native au format Iceberg depuis les moteurs Fabric (pas seulement lecture)
- Support des branches et tags Iceberg pour les scénarios de versionnement
- Intégration approfondie avec Apache XTable pour la conversion omni-directionnelle automatique

Long terme :

- Convergence possible des formats de table vers une spécification unifiée
- Intelligence artificielle intégrée pour l'optimisation automatique des performances

Les architectes doivent suivre ces évolutions et planifier leur architecture pour bénéficier des nouvelles fonctionnalités dès leur disponibilité.

Résumé

Ce chapitre a exploré en profondeur l'intégration entre Apache Iceberg et Microsoft Fabric, une convergence qui illustre la tendance vers l'interopérabilité des formats de table ouverts dans l'écosystème du Data Lakehouse moderne.

Points clés à retenir :

OneLake et la virtualisation des métadonnées. OneLake offre un lac de données logique unifié pour l'ensemble de l'organisation, éliminant les silos traditionnels entre les différentes charges de travail analytiques. Les raccourcis permettent d'accéder aux données stockées dans divers emplacements — Amazon S3, Google Cloud Storage, Azure Data Lake Storage, ou même des sources sur site — sans duplication physique des données. La virtualisation des métadonnées, basée sur Apache XTable, permet aux tables Iceberg d'être interprétées comme Delta Lake et vice versa, offrant une interopérabilité bidirectionnelle complète sans conversion des fichiers de données Parquet sous-jacents.

Power BI Direct Lake. Ce mode de stockage révolutionnaire, exclusif à Microsoft Fabric, combine les performances exceptionnelles du mode Import avec la fraîcheur des données du mode DirectQuery. En chargeant directement les fichiers Parquet en mémoire via le moteur VertiPaq, Direct Lake élimine le besoin d'actualisations planifiées complexes et coûteuses tout en offrant des performances de requête optimales pour les utilisateurs finaux. L'optimisation V-Order maximise ces performances pour les tables Delta natives, tandis que les tables Iceberg virtualisées via raccourcis offrent des performances légèrement réduites mais toujours excellentes pour la majorité des cas d'usage analytiques.

Partenariat stratégique Microsoft-Snowflake. L'intégration approfondie entre Microsoft Fabric et Snowflake, basée sur le format ouvert Apache Iceberg, permet aux organisations d'utiliser le meilleur outil pour chaque tâche. Les data scientists peuvent utiliser Snowpark pour le machine learning, les ingénieurs de données peuvent exploiter les capacités de Snowflake pour le traitement de données complexes, tandis que les analystes métier créent des rapports et tableaux de bord dans Power BI — le tout sur une copie unique des données, sans duplication ni pipelines de synchronisation complexes.

Agents de données et intelligence artificielle. L'intégration native de Fabric avec Copilot et les agents de données ouvre de nouvelles possibilités pour l'analyse conversationnelle. Les utilisateurs peuvent interroger leurs données Iceberg en langage naturel, démocratisant l'accès aux insights analytiques au-delà des équipes techniques traditionnelles.

Patterns d'architecture. Trois patterns principaux ont été présentés pour guider les architectes dans leurs choix de conception :

1. Le **Lakehouse hybride multi-format** convient aux organisations avec des investissements Iceberg significatifs, permettant une coexistence progressive des deux formats.
2. **Fabric comme couche de consommation unifiée** offre une intégration légère où Iceberg reste le format principal pour le stockage et l'ingestion.
3. La **migration progressive** vers Fabric natif suit une approche en trois phases pour les organisations souhaitant consolider sur une plateforme unique.

Considérations canadiennes. L'intégration Fabric-Iceberg répond particulièrement bien aux besoins spécifiques des organisations canadiennes. La résidence des données est assurée via les régions Azure canadiennes (Canada Central et Canada East), garantissant que les données sensibles demeurent sur le territoire canadien conformément aux exigences réglementaires. La gouvernance unifiée de OneLake facilite la conformité avec les réglementations comme LPRPDE et les exigences sectorielles spécifiques (BSIF pour les institutions financières, par exemple). La flexibilité multi-infonuagique permet aux organisations opérant dans des environnements hybrides de maintenir leur infrastructure existante tout en bénéficiant des capacités analytiques avancées de Fabric.

Recommandations clés pour les architectes :

1. **Évaluer les coûts de sortie (egress)** avant de choisir entre raccourcis et migration vers OneLake. Pour les tables fréquemment accédées depuis des fournisseurs infonuagiques tiers, les coûts de sortie récurrents peuvent dépasser rapidement le coût de stockage dans OneLake.

2. **Optimiser les fichiers Parquet sources** pour maximiser les performances Direct Lake. Le tri par les colonnes de filtrage courantes, la compaction régulière, et le partitionnement stratégique améliorent significativement les temps de réponse.
3. **Appliquer la sécurité OneLake** pour une gouvernance unifiée sur les données virtualisées. Même si les tables Iceberg sources ne disposent pas de contrôles d'accès granulaires, OneLake permet d'appliquer des restrictions au niveau des dossiers, des lignes et des colonnes.
4. **Surveiller les métriques de performance** pour identifier les optimisations nécessaires. Les températures de colonnes, les taux de repli DirectQuery, et les temps de conversion des métadonnées sont des indicateurs clés à suivre.
5. **Planifier la migration progressive** plutôt qu'un basculement complet pour minimiser les risques et permettre une validation incrémentielle des fonctionnalités.
6. **Documenter le lignage des données** depuis les sources Iceberg jusqu'aux rapports Power BI pour faciliter la gouvernance et le dépannage.
7. **Prévoir la continuité d'activité** en identifiant les points de défaillance potentiels et en implémentant des stratégies de résilience appropriées.

Perspective stratégique :

L'intégration de Microsoft Fabric avec Apache Iceberg représente une évolution majeure vers l'interopérabilité dans l'écosystème des données d'entreprise. Cette convergence libère les organisations de la dépendance à un seul fournisseur ou format, permettant de choisir les meilleurs outils pour chaque tâche tout en maintenant une vue unifiée des données. Les organisations peuvent désormais bénéficier simultanément de la flexibilité et de l'ouverture d'Apache Iceberg, de la puissance analytique de Power BI Direct Lake, de l'intégration native avec Microsoft 365 et Copilot, et de la gouvernance centralisée de OneLake.

Cette convergence positionne les architectes de données pour construire des plateformes analytiques modernes qui sont à la fois performantes, ouvertes et prêtes pour l'ère de l'intelligence artificielle. Les investissements dans Apache Iceberg ne sont plus en opposition avec l'adoption de Microsoft Fabric — ils sont complémentaires, permettant aux organisations de tirer le meilleur parti des deux écosystèmes.

⬅️ Précédent	🏠 Sommaire	➡️ Suivant
Chapitre IV.13 - Sécurité et Gouvernance	Table des matières	Chapitre IV.15 - Contexte Canadien et Études de Cas

Chapitre IV.15 — Contexte Canadien et Études de Cas

Partie 4 : Intégrations et Perspectives

Introduction

Le Canada occupe une position unique dans l'écosystème mondial des données. Deuxième pays en superficie mondiale, il combine une économie développée du G7 avec des défis géographiques majeurs : une population dispersée sur un vaste territoire, des industries primaires fortement dépendantes des données (ressources naturelles, énergie, agriculture) et une proximité économique intense avec les États-Unis qui soulève des questions fondamentales de souveraineté numérique.

Pour les architectes data canadiens, la conception d'un Data Lakehouse ne peut ignorer ce contexte singulier. Les réglementations provinciales divergentes, notamment la Loi 25 du Québec qui représente l'une des législations de protection des données les plus strictes en Amérique du Nord, les exigences du Bureau du surintendant des institutions financières (BSIF) pour le secteur bancaire et les considérations de résidence des données imposent des contraintes architecturales spécifiques que les modèles génériques ne peuvent pas adresser.

Ce chapitre examine le contexte canadien sous l'angle du Data Lakehouse : nous analysons d'abord l'écosystème réglementaire et infrastructurel canadien, puis explorons deux études de cas majeures — la Banque Royale du Canada (RBC) et Bell Canada — qui illustrent comment des organisations de premier plan ont modernisé leurs plateformes de données. Nous concluons par une analyse approfondie des enjeux de souveraineté des données et de l'infrastructure régionale disponible pour les architectes canadiens.

IV.15.1 Introduction au Contexte Canadien

Le Paysage Numérique Canadien

Le Canada a connu une accélération remarquable de sa transformation numérique au cours des dernières années. Selon les données récentes, 85 % des organisations canadiennes utiliseront des environnements hybrides combinant infrastructure locale, nuage privé et nuage public d'ici la fin de 2025, comparativement à 60 % en 2021. Cette transition massive vers l'infonuagique s'accompagne d'une croissance exponentielle des volumes de données et d'une adoption accélérée de l'intelligence artificielle (IA).

Le marché canadien de la transformation numérique présente des caractéristiques distinctives. Le secteur des services bancaires et financiers (BFSI) représente 24 % des dépenses en 2024, suivi par le secteur de la santé qui affiche la croissance la plus rapide avec un taux composé de 29 %. L'Ontario domine avec 37 % du marché, tandis que l'Alberta affiche la croissance la plus dynamique à 28,6 % de taux composé annuel.

Cette effervescence numérique crée un contexte favorable pour l'adoption du Data Lakehouse comme architecture de référence. Les organisations canadiennes cherchent à unifier leurs données dispersées, à réduire les coûts d'infrastructure et à accélérer leurs initiatives d'IA et d'analytique avancée. Apache Iceberg, avec son approche ouverte et son support multi-moteur, répond particulièrement bien à ces exigences dans un environnement où la flexibilité et l'évitement de la dépendance à un fournisseur unique sont des priorités stratégiques.

Cadre Réglementaire et Conformité

Le paysage réglementaire canadien en matière de données se caractérise par une superposition de juridictions fédérales et provinciales qui crée une complexité significative pour les architectes data.

La LPRPDE (PIPEDA) au Niveau Fédéral

La Loi sur la protection des renseignements personnels et les documents électroniques (LPRPDE) constitue le cadre fédéral régissant la collecte, l'utilisation et la divulgation des renseignements personnels dans le secteur privé. Entrée en vigueur en 2001 et mise à jour périodiquement, elle s'applique aux organisations qui recueillent, utilisent ou communiquent des renseignements personnels dans le cadre d'activités commerciales.

Les dix principes fondamentaux de la LPRPDE guident la conception des systèmes de données :

1. **Responsabilité** : L'organisation est responsable des renseignements personnels dont elle a la gestion et doit désigner une personne chargée d'assurer le respect des principes.
2. **Détermination des fins** : Les fins auxquelles les renseignements personnels sont recueillis doivent être déterminées avant ou au moment de la collecte.
3. **Consentement** : Toute personne doit être informée de toute collecte, utilisation ou communication de renseignements personnels et y consentir.
4. **Limitation de la collecte** : Seuls les renseignements nécessaires aux fins déterminées doivent être recueillis.
5. **Limitation de l'utilisation, de la communication et de la conservation** : Les renseignements ne doivent être utilisés ou communiqués qu'aux fins pour lesquelles ils ont été recueillis, à moins que la personne n'y consente.
6. **Exactitude** : Les renseignements personnels doivent être aussi exacts, complets et à jour que possible.
7. **Mesures de sécurité** : Les renseignements personnels doivent être protégés par des mesures de sécurité appropriées.
8. **Transparence** : Une organisation doit rendre facilement accessible son information relative à ses politiques et pratiques.
9. **Accès aux renseignements personnels** : Toute personne doit pouvoir consulter les renseignements personnels la concernant et les faire rectifier.
10. **Possibilité de porter plainte** : Toute personne doit pouvoir se plaindre du non-respect de ces principes.

Contrairement aux idées reçues, la LPRPDE n'exige pas explicitement que les données soient stockées au Canada. Elle impose plutôt que les organisations protègent les renseignements personnels, quel que soit leur emplacement de stockage. Cette approche basée sur les principes plutôt que sur les règles

prescriptives offre une flexibilité aux organisations mais crée également une ambiguïté que la Loi 25 du Québec a cherché à clarifier.

Cependant, cette flexibilité apparente se heurte à des considérations pratiques. Le transfert de données vers des juridictions étrangères implique que ces données deviennent potentiellement assujetties aux lois de ces juridictions. La Clarifying Lawful Overseas Use of Data Act (CLOUD Act) américaine de 2018, par exemple, permet aux autorités américaines d'exiger l'accès aux données contrôlées par des entreprises américaines, même si ces données résident physiquement au Canada.

Pour les architectes lakehouse, la LPRPDE impose la mise en place de contrôles d'accès robustes, de mécanismes d'audit permettant de démontrer la conformité, et de processus pour répondre aux demandes d'accès des individus. Apache Iceberg facilite ces exigences grâce à ses capacités de time travel, qui permettent de reconstituer l'état exact des données à un moment donné, et à ses tables de métadonnées qui documentent l'historique complet des opérations sur les données.

La Loi 25 du Québec

La Loi 25 (anciennement projet de loi 64), entrée en vigueur progressivement entre 2022 et 2024, représente une modernisation majeure du cadre de protection des données au Québec. Elle s'inspire du Règlement général sur la protection des données (RGPD) européen et impose des obligations significativement plus strictes que la LPRPDE.

Les principales exigences de la Loi 25 pertinentes pour les architectes lakehouse comprennent :

Exigence	Implication Architecturale
Évaluation des facteurs relatifs à la vie privée (EFVP)	Obligatoire avant tout projet impliquant des données personnelles
Responsable de la protection des renseignements personnels	Désignation obligatoire, défaut au plus haut dirigeant
Droit à la portabilité des données	Format structuré et couramment utilisé requis
Consentement explicite	Mécanismes de consentement granulaire à implémenter
Notification des incidents	Délai de notification strict à l'autorité et aux personnes concernées
Évaluations transfrontalières	Analyse obligatoire avant tout transfert hors Québec

Pour un Data Lakehouse, la Loi 25 impose une traçabilité complète du lignage des données personnelles, des mécanismes d'anonymisation ou de pseudonymisation robustes, et la capacité de répondre aux demandes d'accès et de portabilité dans un délai de 30 jours. Apache Iceberg facilite plusieurs de ces exigences grâce à ses capacités de time travel (permettant de reconstituer l'état des données à un moment donné) et ses tables de métadonnées qui documentent l'historique des opérations.

Les Exigences du BSIF pour le Secteur Financier

Le Bureau du surintendant des institutions financières (BSIF) régit les institutions financières fédérales avec des lignes directrices spécifiques concernant la technologie et les données. La ligne directrice B-13 sur la gestion du risque technologique et cybernétique, dont la conformité complète est requise d'ici

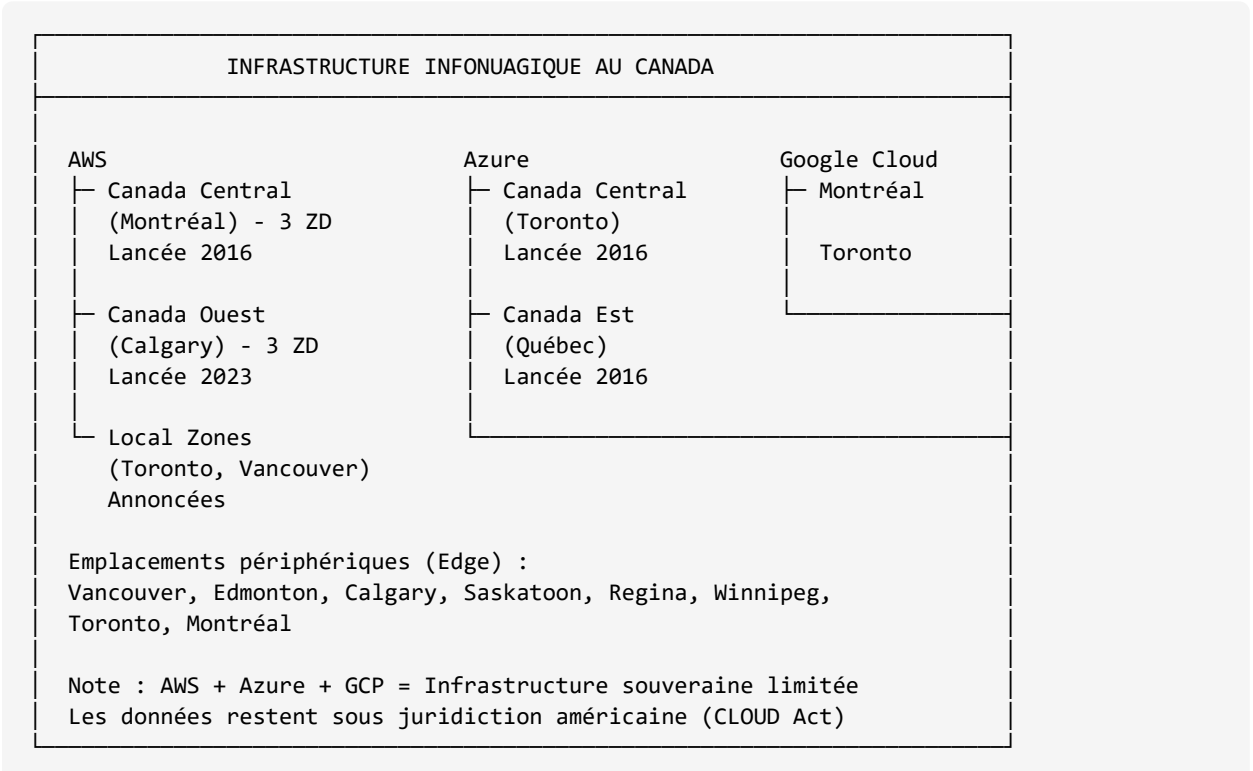
septembre 2026, impose des exigences rigoureuses en matière de résilience opérationnelle et de gestion des tiers.

L'initiative de modernisation de la collecte de données (DCM) lancée par le BSIF en collaboration avec la Banque du Canada et la Société d'assurance-dépôts du Canada représente une transformation majeure. Cette initiative vise à collecter des données réglementaires plus pertinentes, plus ponctuelles et de meilleure qualité grâce à une plateforme technologique moderne. En mars 2025, Regnology a été sélectionné comme fournisseur logiciel pour cette nouvelle plateforme de déclaration réglementaire.

Ces exigences influencent directement l'architecture des lakehouse dans le secteur financier canadien. Les institutions doivent maintenir des pistes d'audit complètes, démontrer leur capacité de récupération après sinistre et assurer la continuité de leurs opérations critiques même en cas de défaillance de fournisseurs tiers.

Infrastructure Infonuagique au Canada

L'infrastructure infonuagique disponible au Canada a considérablement mûri au cours des dernières années. Les trois grands fournisseurs hyperscale disposent tous de régions canadiennes :



Cette infrastructure permet aux organisations canadiennes de stocker et traiter leurs données sur le sol canadien, satisfaisant ainsi les exigences de résidence des données. Cependant, comme nous l'explorerons dans la section sur la souveraineté, la résidence ne garantit pas la souveraineté — une distinction cruciale que les architectes doivent comprendre.

Au-delà des hyperscalers, le Canada dispose d'un écosystème de centres de données locaux en croissance. Des opérateurs comme Cologix, eStruxture, Equinix et Vantage offrent des services de colocation dans les principales métropoles. Bell Canada a annoncé en 2025 le développement de six centres de données en Colombie-Britannique avec une capacité totale de 500 mégawatts, positionnant l'entreprise comme un acteur majeur de l'infrastructure IA au pays.

IV.15.2 Étude de Cas : Banque Royale du Canada (RBC)

Étude de cas : Banque Royale du Canada
Secteur : Services financiers
Défi : Moderniser l'infrastructure data pour supporter l'IA à grande échelle et la conformité réglementaire
Solution : Architecture événementielle avec Confluent Platform, nuage privé IA avec Red Hat et NVIDIA, plateforme infonuagique hybride
Résultats : Classement #3 mondial et #1 au Canada pour la maturité IA (Evident AI Index 2024), analyse de 11 billions d'événements de sécurité en 2024

Contexte et Vision Stratégique

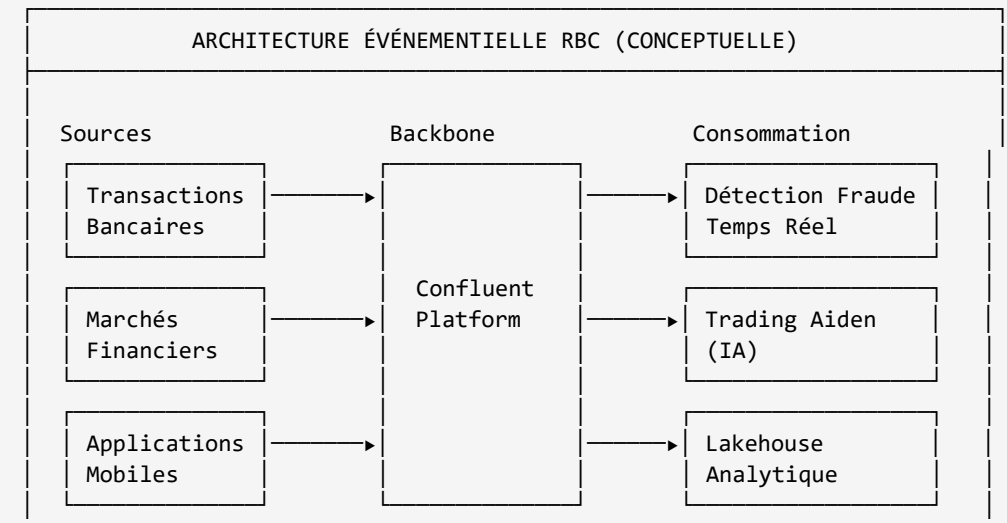
La Banque Royale du Canada (RBC) est la plus grande banque du pays par capitalisation boursière et figure parmi les dix plus grandes banques mondiales. Avec plus de 98 000 employés, des opérations dans 29 pays et un revenu annuel total de 57 milliards de dollars en 2024, RBC représente un cas d'étude exemplaire pour comprendre comment une institution financière de premier plan aborde la modernisation de ses données.

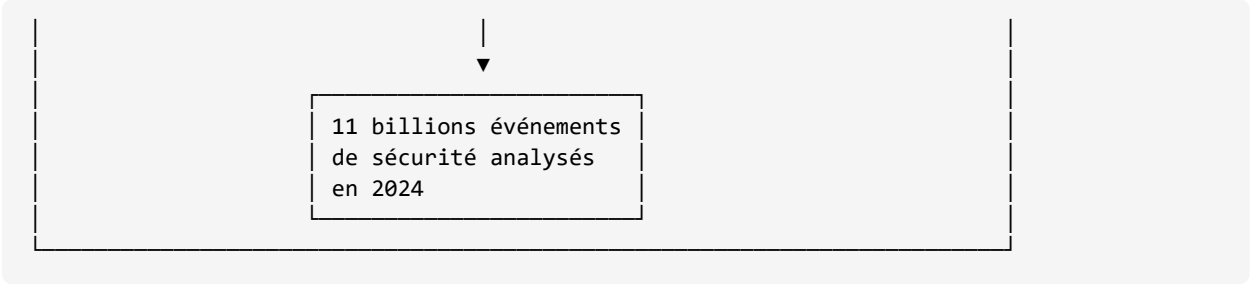
La stratégie technologique de RBC repose sur trois piliers fondamentaux : l'investissement massif dans l'IA et l'apprentissage automatique via Borealis AI, l'adoption d'architectures infonuagiques natives et hybrides, et la mise en place d'une infrastructure événementielle à grande échelle. En 2025, la banque investit plus de 5 milliards de dollars en technologie, démontrant l'ampleur de son engagement.

Architecture de Données Événementielle avec Confluent

RBC a sélectionné Confluent Platform comme fondation de son architecture de données événementielle. Cette décision stratégique répond à plusieurs impératifs : supporter le nombre croissant d'initiatives d'IA et d'apprentissage automatique, créer une architecture évolutive et pilotée par les événements, et établir une plateforme de données en temps réel pour l'ensemble de l'organisation.

L'architecture événementielle de RBC illustre le pattern du Streaming Lakehouse que nous avons exploré au Chapitre IV.12. Apache Kafka sert de backbone pour les flux de données en temps réel, tandis que les données historiques sont persistées dans des formats de table ouverts pour l'analytique avancée.





Cette architecture permet à RBC de traiter des volumes massifs de données en temps réel. En 2024, la plateforme a analysé environ 11 billions d'événements de sécurité, alimentant les systèmes de détection de fraude et de gestion des risques de la banque.

Borealis AI et l'Infrastructure IA Privée

Borealis AI, l'institut de recherche en IA de RBC fondé en 2016, constitue le centre d'excellence de la banque pour l'intelligence artificielle. Avec plus de 950 employés et des laboratoires à Toronto, Montréal, Waterloo, Vancouver et Calgary, Borealis AI représente l'un des plus importants investissements en recherche IA d'une institution financière canadienne.

En 2020, RBC et Borealis AI ont annoncé le développement d'une plateforme de nuage privé IA en partenariat avec Red Hat et NVIDIA. Cette infrastructure utilise des systèmes NVIDIA DGX orchestrés par Red Hat OpenShift, permettant de construire, déployer et maintenir des applications bancaires de nouvelle génération alimentées par l'IA.

> **Performance**
> La plateforme de nuage privé IA de RBC peut exécuter des milliers de simulations et analyser des millions de points de données en une fraction du temps comparativement aux systèmes traditionnels. Cette capacité a amélioré l'exécution des transactions boursières, réduit les appels clients grâce à des interactions plus intelligentes et accéléré la livraison de nouvelles applications.

L'architecture de Borealis AI repose sur Red Hat Enterprise Linux (RHEL) comme fondation et Red Hat OpenShift comme plateforme d'orchestration des conteneurs. Cette combinaison permet une collaboration efficace entre scientifiques de données, ingénieurs de données et développeurs logiciels, accélérant le déploiement des modèles d'apprentissage automatique et d'apprentissage profond en production.

Aiden : La Plateforme de Trading Algorithmique

Un exemple concret de l'application de l'IA chez RBC est Aiden, la plateforme de trading électronique alimentée par l'IA lancée en 2020. Aiden utilise l'apprentissage par renforcement profond pour apprendre en temps réel sur le trading d'actions et exécuter automatiquement des transactions d'achat et de vente dans le meilleur intérêt des clients.

La plateforme anticipe les changements du marché boursier et ajuste les exécutions de transactions en conséquence. Elle aide les clients de RBC à prendre des décisions d'investissement plus éclairées tout en optimisant les coûts d'exécution. Cette application illustre comment un lakehouse alimenté par des flux en temps réel peut supporter des cas d'utilisation à faible latence et à haute valeur ajoutée.

Initiative de Modernisation de la Fraude

Face à l'évolution des menaces cybernétiques, RBC a lancé une initiative de modernisation de la fraude qui représente une refonte stratégique de sa posture de sécurité. Cette initiative marque la transition des systèmes traditionnels basés sur des règles statiques vers des moteurs de notation des risques adaptatifs et en temps réel alimentés par l'IA et l'apprentissage automatique.

Cette modernisation a été reconnue par un prix CIO 2024 d'IDC et Foundry. L'infrastructure fondamentale a été construite sur mesure, permettant le traitement d'événements complexes et intégrant des capacités d'analyse comportementale et de prédiction de fraude.

Reconnaissance et Maturité IA

La validation la plus puissante de la stratégie IA de RBC provient de l'Evident AI Index, un benchmark indépendant et basé sur les données qui évalue la maturité IA des 50 plus grandes institutions financières mondiales. Dans l'indice d'octobre 2024, RBC s'est classée #3 au niveau mondial et #1 au Canada pour la maturité IA globale, maintenant sa position dans le top trois mondial pour la troisième année consécutive.

Critère	Classement RBC	Note
Maturité IA globale	#3 mondial, #1 Canada	Troisième année consécutive
Pilier Talent	Top 10 mondial	Excellence en acquisition de talents IA
Pilier Innovation	Top 10 mondial	Plateforme Aiden, recherche Borealis
Pilier Leadership	Top 10 mondial	Engagement de la direction
Pilier Transparence	Top 10 mondial	Communication des initiatives IA

RBC est l'une des deux seules banques au monde à se classer dans le top 10 des quatre catégories, démontrant une stratégie IA bien équilibrée et profondément intégrée.

Leçons pour les Architectes Lakehouse

L'expérience de RBC offre plusieurs enseignements pour les architectes concevant des lakehouse dans le secteur financier canadien :

L'architecture événementielle comme fondation est essentielle. Le choix de Confluent Platform comme backbone permet d'unifier les flux de données en temps réel et les charges de travail analytiques, un pattern que le Streaming Lakehouse avec Apache Iceberg peut adresser efficacement. Cette approche élimine la dichotomie traditionnelle entre systèmes opérationnels et systèmes analytiques, permettant une vue unifiée des données.

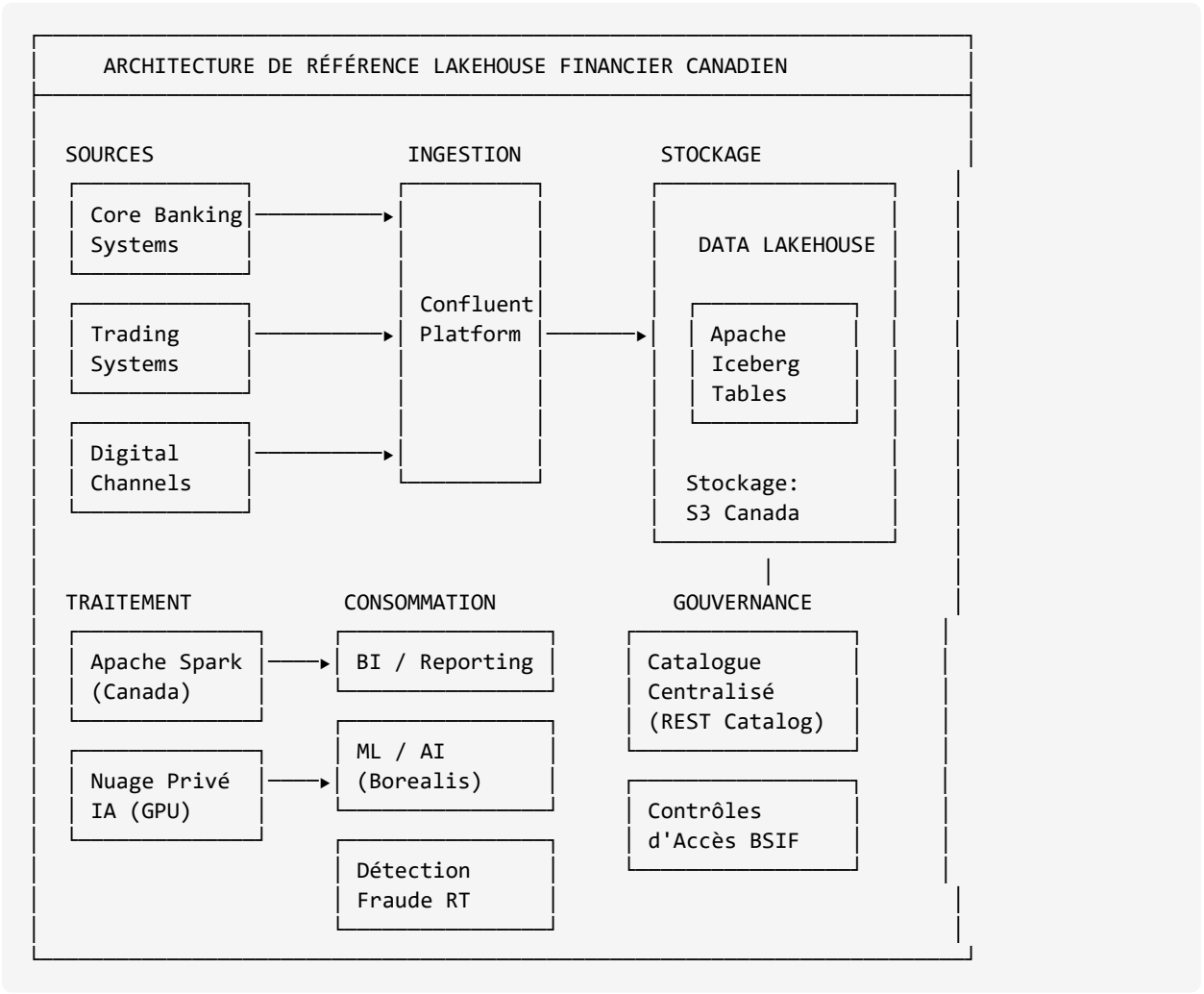
L'investissement dans une infrastructure IA souveraine permet de maintenir le contrôle sur les données sensibles. Le nuage privé IA de RBC démontre qu'une institution financière peut développer des capacités IA de pointe tout en conservant la maîtrise de ses données critiques. Cette approche hybride — infonuagique publique pour les charges de travail générales, infrastructure privée pour l'IA et les données sensibles — représente un modèle reproductible.

La conformité réglementaire doit être intégrée dès la conception. Les exigences du BSIF en matière de résilience opérationnelle, de gestion des tiers et de récupération après sinistre influencent directement l'architecture de la plateforme de données. Apache Iceberg, avec ses snapshots immuables et ses capacités de time travel, offre des mécanismes naturels pour satisfaire ces exigences.

Enfin, l'échelle nécessite une approche industrialisée. Avec 11 billions d'événements de sécurité analysés annuellement, RBC démontre l'importance d'une plateforme capable de traiter des volumes massifs avec des performances prévisibles.

Architecture de Référence Inspirée de RBC

Bien que l'architecture exacte de RBC soit propriétaire, les informations publiques permettent d'esquisser une architecture de référence pour les institutions financières canadiennes :



Les composantes clés de cette architecture incluent :

- **Backbone événementiel** : Confluent Platform ou Apache Kafka géré, déployé dans une région canadienne, servant de système nerveux central pour tous les flux de données.
- **Couche de stockage** : Stockage objet compatible S3 (AWS S3 dans la région Canada Central ou équivalent) avec les données organisées en tables Apache Iceberg.
- **Catalogue centralisé** : REST Catalog ou équivalent commercial (comme Dremio Arctic ou Tabular) pour la découverte des données et la gestion des métadonnées.

- **Moteurs de traitement** : Apache Spark pour les charges de travail batch et le traitement de données à grande échelle, déployé dans des régions canadiennes.
- **Infrastructure IA dédiée** : Clusters GPU pour l'entraînement et l'inférence des modèles d'apprentissage automatique, potentiellement dans un nuage privé pour les cas d'utilisation les plus sensibles.
- **Contrôles de gouvernance** : Intégration avec les systèmes d'identité de l'entreprise, audit complet des accès, et politiques alignées avec les exigences du BSIF.

IV.15.3 Étude de Cas : Bell Canada

> **Étude de cas : Bell Canada**
> *Secteur* : Télécommunications et médias
> *Défi* : Moderniser l'infrastructure technologique pour supporter les opérations réseau, l'expérience client et la croissance des services IA
> *Solution* : Partenariat stratégique avec Google Cloud, transformation avec ServiceNow, développement de centres de données IA
> *Résultats* : Revenus IA projetés de 700 millions CAD en 2025 avec croissance de 29 % TCAC, amélioration significative du temps moyen de résolution (MTTR) des incidents réseau

Contexte et Transformation Numérique

Bell Canada est la plus grande entreprise de communications du pays avec plus de 22 millions de connexions clients et une histoire de 145 ans d'innovation. L'entreprise opère dans quatre segments : services sans fil, services filaires, Bell Média et Bell Affaires. Cette diversité crée des défis uniques en matière de gestion des données, chaque segment générant des volumes massifs d'information avec des caractéristiques distinctes.

La transformation numérique de Bell repose sur plusieurs initiatives stratégiques parallèles : un partenariat pluriannuel avec Google Cloud pour la modernisation de l'infrastructure, une alliance étendue avec ServiceNow pour l'automatisation des processus et le développement d'une offre de services IA et cybersécurité à destination des entreprises.

Partenariat Stratégique avec Google Cloud

En juillet 2021, Bell et Google Cloud ont annoncé un partenariat stratégique pluriannuel visant à accélérer la transformation numérique de Bell, à améliorer son infrastructure réseau et informatique, et à contribuer à un avenir plus durable. Ce partenariat combine le leadership de Bell dans les réseaux 5G avec l'expertise de Google en matière d'infonuagique multi-cloud, d'analytique de données et d'intelligence artificielle.

Les composantes clés de ce partenariat incluent :

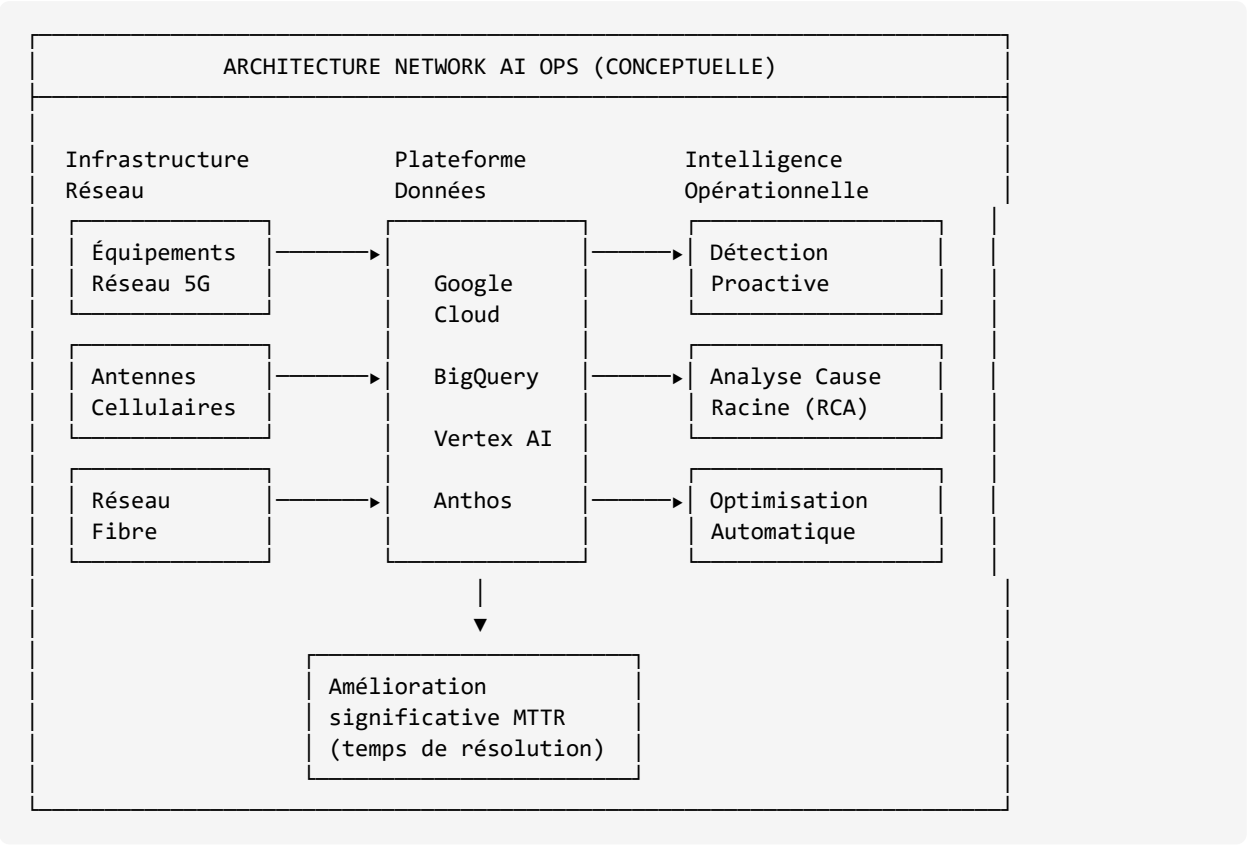
La migration des charges de travail critiques vers le nuage représente un volet fondamental. En déplaçant et modernisant l'infrastructure informatique, les fonctions réseau et les applications critiques de l'infrastructure locale vers Google Cloud, Bell améliore son efficacité opérationnelle et les performances de ses applications.

Le déverrouillage de la technologie réseau multi-cloud de nouvelle génération constitue un autre axe stratégique. Avec la puissance combinée du réseau 5G de Bell et d'Anthos, la solution multi-cloud de Google, Bell peut offrir une expérience client cohérente avec une automatisation accrue et une flexibilité améliorée qui s'adapte à la demande.

L'exploitation de la puissance de l'IA, des données et de l'analytique permet à Bell de tirer parti de l'expertise de Google Cloud pour obtenir des perspectives uniques grâce à l'analytique de données réseau en temps réel, améliorant l'expérience client et l'assurance des services.

Solution Network AI Ops

En février 2025, Bell a annoncé le déploiement réussi de sa solution Network AI Ops construite sur Google Cloud. Cette solution révolutionne la façon dont Bell détecte, analyse et présente les problèmes réseau, marquant le passage d'une approche réactive à une approche proactive de la gestion du réseau.



L'approche pilotée par l'IA a amélioré significativement le temps moyen de résolution (MTTR) de Bell, permettant une gestion proactive du réseau qui améliore la fiabilité et optimise l'expérience client. Cette solution adresse directement les défis d'une ère où les réseaux de télécommunications font face à une complexité croissante et à une explosion du trafic de données.

Transformation avec ServiceNow

En juillet 2024, Bell et ServiceNow ont annoncé un accord stratégique pluriannuel étendu pour accélérer la transformation de Bell. Cet accord fait de Bell l'un des plus grands clients de ServiceNow dans le secteur des communications, avec une collaboration première au Canada.

Bell étend son utilisation de la plateforme ServiceNow pour supporter sa propre transformation numérique tout en continuant à offrir l'expertise d'implémentation ServiceNow pour supporter la

transformation numérique de ses clients Bell Affaires. FX Innovation, un leader des services axés sur l'infonuagique et partenaire Elite d'implémentation ServiceNow acquis par Bell en 2023, implémente la plateforme Now à travers l'écosystème de Bell.

Cette alliance avec ServiceNow s'inscrit dans le partenariat pluriannuel signé en 2025 entre Bell Canada et ServiceNow pour automatiser les flux de travail internes et revendre des solutions packagées aux clients entreprises. Ce type de partenariat illustre comment les télécommunicateurs canadiens étendent leur proposition de valeur au-delà de la connectivité pure vers des services de plateforme et d'automatisation.

Stratégie IA et Centres de Données

Bell a positionné l'IA et l'infrastructure de données comme piliers centraux de sa croissance future. En octobre 2025, le PDG Mirko Bibic a annoncé une cible de revenus IA de 1,07 milliard CAD d'ici 2028, avec des revenus attendus d'environ 700 millions CAD en 2025 et une croissance à un taux composé annuel allant jusqu'à 29 % sur les trois prochaines années.

Bell génère déjà des revenus IA à partir de plusieurs sources : la location de capacité de centre de données, la vente de solutions de cybersécurité et l'intégration de systèmes technologiques via sa division de conseil Ateko. En mai 2025, l'opérateur de télécommunications a annoncé des plans pour développer six centres de données en Colombie-Britannique avec une capacité totale de 500 mégawatts.

À la différence de certains concurrents, Bell n'achètera pas ses propres GPU, préférant louer de l'espace à des entreprises qui le font. Groq, basé en Californie, a déjà loué une grande partie du site de Bell à Kamloops, tandis que Buzz HPC déploie des GPU NVIDIA dans une installation de cinq mégawatts au Manitoba.

Bell Cyber et la Souveraineté des Données

En septembre 2025, Bell a officiellement annoncé le lancement de Bell Cyber lors du premier Sommet Bell sur la cybersécurité à Toronto. Cette nouvelle marque complète l'offre croissante de services technologiques alimentés par l'IA de Bell, aux côtés d'Ateko et de Bell AI Fabric.

Bell Cyber représente une étape cruciale dans la stratégie de Bell pour devenir un chef de file nord-américain des services technologiques, offrant des solutions de cybersécurité évolutives, souveraines et alimentées par l'IA qui protègent les entreprises et renforcent l'avenir numérique du Canada. Cette offre répond à une préoccupation majeure des dirigeants d'entreprises canadiennes : selon une étude commandée par Bell en octobre 2025, neuf dirigeants d'entreprises sur dix affirment qu'il est plus important que jamais de conserver les données sensibles au Canada.

> **Performance**
> Les résultats internes de Bell pour 2023-2024 montrent une réduction des ressources totales nécessaires pour maintenir la plateforme de données d'entreprise aux mêmes niveaux de service. Les tests A/B internes mesurant l'impact des initiatives pilotées par l'IA ont démontré des améliorations significatives de l'expérience client.

Lakehouse pour les Télécommunications

L'expérience de Bell illustre plusieurs patterns pertinents pour les architectes lakehouse dans le secteur des télécommunications :

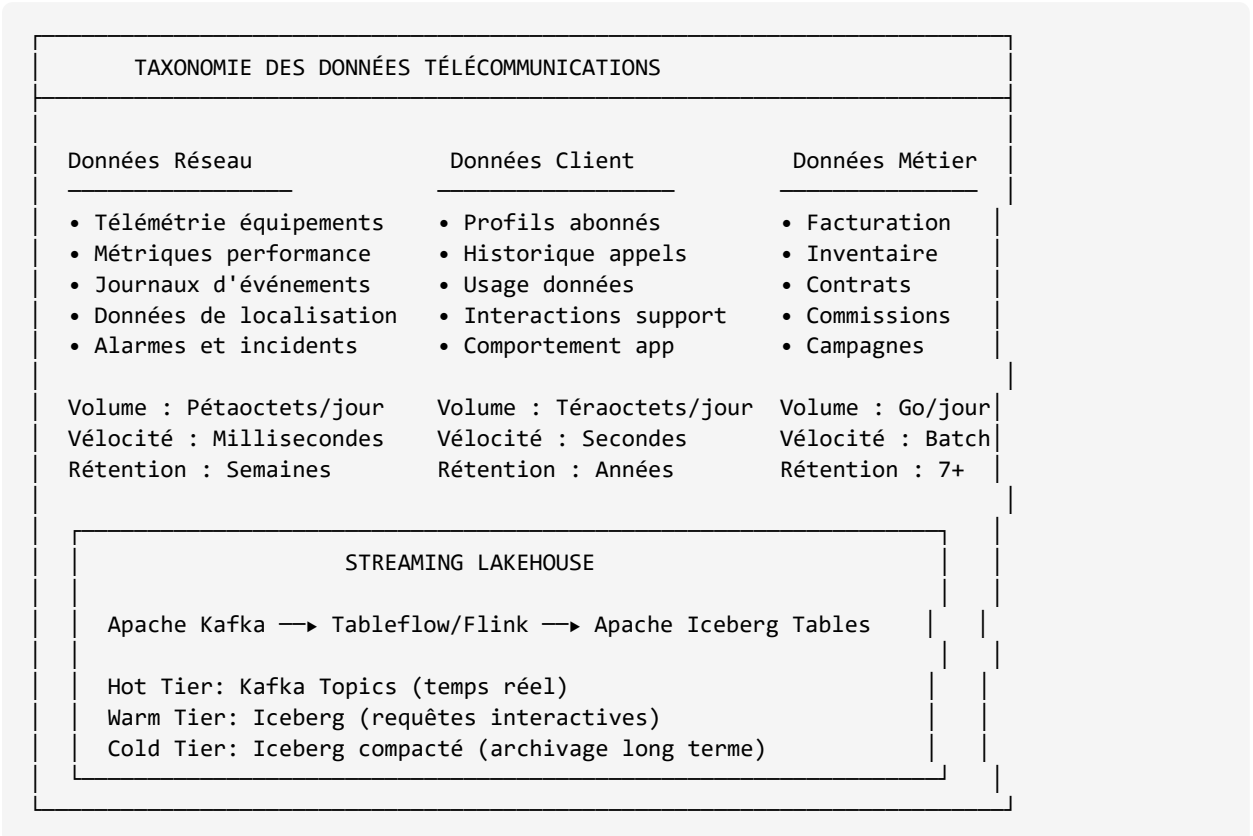
L'analytique réseau en temps réel nécessite une architecture capable de traiter des flux massifs de données de télémétrie et d'événements réseau. Un Streaming Lakehouse avec Apache Iceberg et Apache Kafka représente une architecture naturelle pour ce type de charge de travail, permettant à la fois l'analyse en temps réel et l'interrogation historique sur les mêmes données.

L'expérience client multicanal requiert l'unification des données provenant de multiples points de contact : applications mobiles, centres d'appels, portails web, interactions en magasin. Le Data Lakehouse offre une fondation pour créer une vue 360 degrés du client tout en supportant les exigences de conformité en matière de protection des renseignements personnels.

La monétisation des données représente une opportunité croissante. Les données de réseau, agrégées et anonymisées, peuvent alimenter des services à valeur ajoutée pour les clients entreprises. L'architecture lakehouse avec ses capacités de gouvernance intégrées facilite cette monétisation tout en respectant les cadres réglementaires.

Architecture de Données Télécommunications

Le secteur des télécommunications génère certains des volumes de données les plus importants de toute industrie. Une architecture lakehouse typique pour un télécommunicateur canadien doit adresser plusieurs types de données avec des caractéristiques très différentes :



Les cas d'utilisation analytique dans les télécommunications bénéficient directement de l'architecture lakehouse :

La maintenance prédictive du réseau combine les données historiques de performance des équipements avec les flux de télémétrie en temps réel. Les modèles d'apprentissage automatique peuvent prédire les défaillances avant qu'elles n'impactent les clients. Le lakehouse permet d'entraîner ces modèles sur des années de données historiques tout en les alimentant avec des données en quasi temps réel.

L'optimisation de l'expérience client s'appuie sur l'analyse du comportement à travers tous les canaux pour personnaliser les offres, prédire le désabonnement et optimiser les interactions de service. La vue unifiée offerte par le lakehouse est essentielle pour ces cas d'utilisation.

La détection des fraudes dans les télécommunications (fraude à l'abonnement, fraude à l'itinérance, fraude aux appels premium) nécessite l'analyse de comportements anormaux dans de grands volumes de données. Le lakehouse permet ces analyses à l'échelle requise.

La conformité réglementaire imposée par le CRTC (Conseil de la radiodiffusion et des télécommunications canadiennes) exige des capacités de conservation et de rapport que le lakehouse peut satisfaire grâce à ses capacités de rétention longue durée et de requêtes ad hoc.

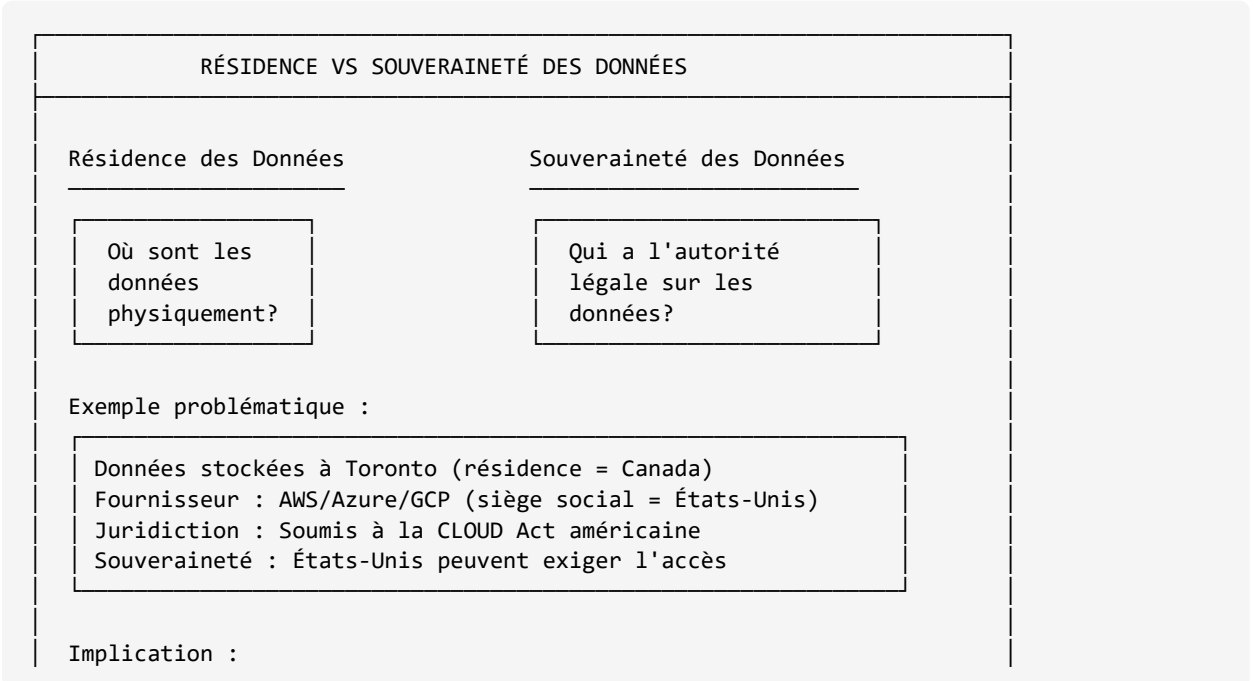
IV.15.4 Souveraineté des Données et Infrastructure Régionale

Résidence versus Souveraineté

La distinction entre résidence des données et souveraineté des données est fondamentale pour les architectes canadiens et souvent mal comprise. Cette confusion peut mener à des décisions architecturales qui satisfont les exigences de conformité apparentes tout en exposant l'organisation à des risques juridictionnels significatifs.

La résidence des données réfère à l'emplacement physique où les données sont stockées — par exemple, un serveur situé à Toronto ou à Montréal. Les fournisseurs de services infonuagiques offrent la capacité d'isoler les données pour qu'elles résident dans une région géographique spécifique, ce qui peut aider à respecter les lois d'une juridiction particulière.

La souveraineté des données, en revanche, réfère à l'autorité légale qui gouverne ces données. Même si vos données résident au Canada, si le fournisseur est basé aux États-Unis ou dans un autre pays étranger, elles peuvent être assujetties aux lois de ce pays — comme la CLOUD Act américaine.



```
┌──────────────────────────────────────────────────────────────────┐
│  Conformité LPRPDE/Loi 25 ✓  mais  Exposition CLOUD Act ⚠️       │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

Le gouvernement du Canada reconnaît cette réalité dans son livre blanc sur la souveraineté des données : tant qu'un fournisseur de services infonuagiques opérant au Canada est assujetti aux lois d'un pays étranger, le Canada n'aura pas la souveraineté complète sur ses données.

La CLOUD Act et ses Implications

La Clarifying Lawful Overseas Use of Data Act (CLOUD Act), adoptée en 2018, permet aux forces de l'ordre américaines d'exiger des fournisseurs infonuagiques basés aux États-Unis qu'ils remettent les données qu'ils contrôlent, indépendamment de l'endroit où ces données résident géographiquement.

Si votre entreprise canadienne utilise un service infonuagique appartenant à une entreprise américaine — comme AWS, Microsoft Azure ou Google Cloud — vos données peuvent être accédées par les autorités américaines sans votre consentement ni votre connaissance. Vous pouvez être pleinement conforme à la LPRPDE, tout en étant exposé à une surveillance légale non canadienne.

Les experts avertissent que le gouvernement américain pourrait imposer des contrôles à l'exportation sur les ressources de calcul IA et infonuagique, perturbant les opérations des entreprises canadiennes du jour au lendemain. Cette dépendance à l'infrastructure étrangère introduit des risques réels pour la souveraineté économique et la sécurité nationale du Canada.

Stratégies d'Atténuation des Risques

Face à ces défis, plusieurs stratégies s'offrent aux architectes canadiens pour atténuer les risques de souveraineté tout en bénéficiant des avantages de l'infonuagique :

L'utilisation de fournisseurs canadiens représente l'option offrant le plus haut niveau de souveraineté. Un fournisseur de services appartenant à des intérêts canadiens, sans opérations ni représentants aux États-Unis ou dans un autre pays étranger, peut être efficace pour atteindre la souveraineté des données, en supposant que les données sont stockées et ne peuvent être accédées qu'au Canada. Cependant, la disponibilité d'outils modernes est limitée avec cette approche.

Le traitement des données exclusivement dans les locaux de l'organisation et derrière son propre pare-feu constitue une deuxième option. Cela isolera probablement les données des mandats étrangers si l'organisation est détenue et contrôlée au Canada et n'a aucune présence étrangère. Cependant, maintenir une infrastructure locale nécessite des investissements significatifs en capital et en ressources humaines spécialisées.

Les clauses contractuelles robustes avec les fournisseurs infonuagiques représentent une approche intermédiaire. Ces mesures incluent des exigences que les fournisseurs notifient les clients des demandes légales étrangères dans des délais spécifiques, des engagements à contester les divulgations excessives et des garanties de localisation des données.

Le chiffrement avec gestion des clés par le client offre une protection technique. Si les données doivent être remises, elles ne peuvent être déchiffrées sans les clés qui sont généralement gérées par l'organisation ou un tiers neutre.

L'architecture de données hybride permet de segmenter les données selon leur sensibilité et d'appliquer des niveaux de protection différenciés :

Classification	Caractéristiques	Stockage Recommandé	Exemple
Critique	Données réglementées, renseignements personnels sensibles	Infrastructure souveraine ou locale	Données bancaires, dossiers médicaux
Sensible	Renseignements personnels standards, données d'affaires confidentielles	Infonuagique avec résidence canadienne	Profils clients, transactions
Interne	Données opérationnelles non sensibles	Infonuagique standard avec région canadienne	Journaux système, métriques
Public	Données destinées à la publication	Toute infrastructure	Contenus marketing, rapports annuels

Patterns d'Architecture pour les Autres Secteurs Canadiens

Au-delà des services financiers et des télécommunications, plusieurs secteurs canadiens présentent des considérations uniques pour l'architecture lakehouse :

Secteur de l'Énergie et des Ressources Naturelles

Le Canada est un leader mondial dans les secteurs pétrolier, gazier, minier et hydroélectrique. Ces industries génèrent des volumes massifs de données de capteurs IoT, de données géospatiales et de données opérationnelles qui se prêtent naturellement à l'architecture lakehouse.

Les entreprises énergétiques canadiennes comme Hydro-Québec, Suncor et Canadian Natural Resources font face à des défis spécifiques :

- **Données géospatiales massives** : Les données sismiques, les relevés géologiques et les images satellites nécessitent un stockage évolutif et des capacités de traitement parallèle que le lakehouse offre naturellement.
- **Données de capteurs en temps réel** : Les pipelines, les installations de forage et les centrales électriques génèrent des flux continus de télémétrie. Le Streaming Lakehouse avec Apache Kafka et Iceberg permet l'analyse en temps réel et historique.
- **Conformité environnementale** : Les réglementations environnementales exigent une traçabilité complète des données d'émissions et d'impacts. Les capacités de time travel d'Iceberg facilitent les audits réglementaires.
- **Sites distants** : De nombreuses opérations se situent dans des régions éloignées avec une connectivité limitée. Les architectures edge-to-lakehouse permettent le traitement local avec synchronisation différée.

> **Étude de cas : Hydro-Québec (Contexte sectoriel)**
> *Secteur* : Énergie hydroélectrique
> *Défi* : Gérer les données de production de 63 centrales hydroélectriques et d'un réseau de transport

> couvrant plus de 34 000 km
> *Approche typique* : Architecture lakehouse pour unifier les données de production, de maintenance prédictive et de prévision de la demande
> *Considérations* : Société d'État québécoise, données doivent rester au Québec pour des raisons de souveraineté provinciale

Secteur de la Santé

Le système de santé canadien, principalement public et administré par les provinces, présente des défis uniques pour la gestion des données. Les renseignements de santé sont parmi les plus sensibles et les plus réglementés.

Les considérations clés pour les architectures lakehouse en santé incluent :

- **Législations provinciales distinctes** : Chaque province a sa propre législation sur la protection des renseignements de santé (PHIPA en Ontario, par exemple).
- **Interopérabilité FHIR** : Les standards HL7 FHIR (Fast Healthcare Interoperability Resources) définissent des formats d'échange que le lakehouse doit pouvoir ingérer et exposer.
- **Anonymisation pour la recherche** : Les données de santé sont précieuses pour la recherche, mais doivent être anonymisées ou pseudonymisées. Apache Iceberg supporte les vues qui peuvent appliquer des transformations d'anonymisation.
- **Traçabilité des accès** : Les audits d'accès aux dossiers de santé sont légalement requis. Les tables de métadonnées Iceberg peuvent servir de source pour les rapports d'audit.

Secteur Public et Gouvernemental

Les gouvernements fédéral et provinciaux canadiens sont parmi les plus grands producteurs et consommateurs de données au pays. La Direction du gouvernement du Canada sur la résidence des données électroniques établit des exigences claires pour les données aux niveaux Protégé B, Protégé C et Classifié.

Pour les projets de lakehouse impliquant des données gouvernementales :

- Les données Protégé B et supérieures doivent résider au Canada
- Les fournisseurs doivent être évalués selon les certifications reconnues (ISO 27001, SOC 2)
- Le Profil de contrôle de sécurité du GC pour les services TI infonuagiques croise les exigences du GC avec les certifications de l'industrie
- Les initiatives comme le Programme de modernisation de la gestion des données du BSIF influencent les approches de déclaration réglementaire

Infrastructure Régionale Disponible

Le Canada dispose maintenant d'une infrastructure infonuagique substantielle permettant aux organisations de maintenir leurs données sur le sol canadien. Le tableau suivant résume les options principales :

Fournisseur	Régions Canadiennes	Zones de Disponibilité	Notes
AWS	Canada Central (Montréal), Canada Ouest (Calgary)	3 par région	Local Zones annoncées à Toronto et Vancouver
Microsoft Azure	Canada Central (Toronto), Canada Est (Québec)	Non spécifié publiquement	Disponible depuis 2016
Google Cloud	Montréal, Toronto	Non spécifié publiquement	Expansion continue
Oracle Cloud	Canada Sud-Est (Montréal), Canada Sud-Est (Toronto)	1 par région	Oracle Database@Google Cloud disponible
IBM Cloud	Montréal, Toronto	Non spécifié publiquement	Focus entreprise

Au-delà des hyperscalers, des fournisseurs canadiens et internationaux de colocation offrent des alternatives. Cologix, eStruxture, Equinix et Vantage disposent d'installations dans les principales métropoles canadiennes. Ces options peuvent être combinées avec des solutions infonuagiques souveraines ou utilisées pour des charges de travail nécessitant un contrôle maximal.

Considérations pour le Data Lakehouse

Pour un Data Lakehouse basé sur Apache Iceberg, plusieurs considérations spécifiques à la souveraineté doivent être prises en compte :

Le stockage des données représente la couche la plus sensible. Les fichiers Parquet contenant les données réelles doivent être stockés dans des buckets S3 (ou équivalents) situés dans les régions canadiennes. Apache Iceberg est agnostique au stockage, permettant d'utiliser n'importe quel stockage objet compatible S3.

Les métadonnées et le catalogue constituent également des données sensibles qui doivent être protégées. Le REST Catalog d'Iceberg peut être déployé dans une région canadienne, et les métadonnées Iceberg (fichiers manifest, fichiers metadata.json) doivent être colocalisées avec les données.

Le traitement des requêtes peut impliquer le transfert de données entre régions si les moteurs de requête sont déployés dans des juridictions différentes. Les architectes doivent s'assurer que les clusters Apache Spark, Trino ou Dremio traitant les données sensibles sont également déployés dans des régions canadiennes.

Migration
De : Data warehouse infonuagique sans contrôle de résidence
Vers : Data Lakehouse avec stockage souverain
Stratégie :
1. Inventorier toutes les données et leur classification de sensibilité
2. Identifier les données nécessitant une résidence canadienne
3. Déployer l'infrastructure Iceberg dans les régions canadiennes
4. Migrer les données sensibles en premier avec validation de conformité
5. Établir des politiques empêchant l'écriture hors des régions autorisées

Le Cas des Environnements Multi-Cloud et Hybrides

De nombreuses organisations canadiennes opèrent des environnements multi-cloud ou hybrides, combinant plusieurs fournisseurs infonuagiques avec une infrastructure locale. Cette complexité amplifie les défis de souveraineté mais offre également des opportunités d'optimisation.

Apache Iceberg excelle dans ce contexte grâce à son architecture ouverte. Un lakehouse Iceberg peut être déployé de manière à ce que les données résident dans un stockage souverain (par exemple, un stockage objet canadien) tout en étant accessibles par des moteurs de requête déployés dans différents environnements. La couche de métadonnées Iceberg assure la cohérence des vues sur les données indépendamment de l'emplacement des consommateurs.

Cette flexibilité permet des architectures où : - Les données sensibles réglementées résident dans un stockage souverain canadien - Les données moins sensibles peuvent être distribuées globalement pour des raisons de performance - Les moteurs de requête peuvent être déployés près des utilisateurs tout en accédant aux données centralisées - Les politiques de gouvernance sont appliquées uniformément via le catalogue centralisé

IV.15.5 Recommandations pour les Architectes Canadiens

Considérations Économiques

Avant d'aborder les recommandations architecturales, il est essentiel de comprendre les implications économiques des choix de souveraineté et de déploiement au Canada. Les architectes doivent équilibrer plusieurs facteurs de coûts :

Coûts d'infrastructure comparatifs

Les régions canadiennes des hyperscalers présentent généralement un premium de coûts de 5 à 15 % par rapport aux régions américaines comparables, en raison de facteurs comme la capacité plus limitée, les coûts énergétiques et la densité d'infrastructure moindre. Cependant, ce premium doit être mis en balance avec :

- Les coûts de conformité évités en maintenant les données au Canada
- Les risques juridiques et de réputation liés aux violations de souveraineté
- Les coûts de transfert de données (egress) pour les données traversant les frontières

Facteur de Coût	Considération	Impact Typique
Infrastructure infonuagique	Premium régions canadiennes	+5-15% vs US
Transfert de données	Egress transfrontalier vs intra-région	Variable selon volume
Conformité	Coûts d'audit et de certification	Investissement initial significatif
Personnel spécialisé	Expertise locale en souveraineté/conformité	Salaires compétitifs requis
Infrastructure souveraine	Fournisseurs canadiens vs hyperscalers	Souvent +20-40%

Retour sur investissement de la souveraineté

Pour certaines organisations, particulièrement celles dans des secteurs réglementés comme les services financiers ou la santé, l'investissement dans une architecture souveraine génère un retour sur investissement positif lorsqu'on considère :

- L'évitement des sanctions réglementaires potentielles
- La protection de la valeur de marque en cas d'incident de données
- L'accès à des contrats gouvernementaux exigeant la souveraineté
- La différenciation concurrentielle auprès de clients sensibles à la protection des données

Principes Directeurs

À la lumière du contexte canadien, des études de cas et des considérations de souveraineté, les architectes concevant des Data Lakehouse au Canada devraient adhérer aux principes suivants :

La conformité dès la conception doit guider toutes les décisions architecturales. Intégrer les exigences de la LPRPDE, de la Loi 25 et des réglementations sectorielles (BSIF pour les institutions financières, par exemple) dès la phase de conception plutôt qu'en remédiation. Cette approche proactive évite les coûts et les risques associés à la mise en conformité rétroactive.

La souveraineté consciente implique de comprendre la distinction entre résidence et souveraineté et de prendre des décisions éclairées basées sur le profil de risque de l'organisation. Pour les données les plus sensibles, privilégier les options offrant une souveraineté maximale. Pour les données moins sensibles, un équilibre pragmatique peut être approprié.

L'ouverture et la portabilité doivent être favorisées pour éviter la dépendance à un fournisseur unique. Apache Iceberg et les formats ouverts permettent de maintenir la flexibilité de migrer entre fournisseurs si nécessaire. Cette approche est particulièrement pertinente dans un contexte géopolitique incertain où les règles du jeu peuvent changer.

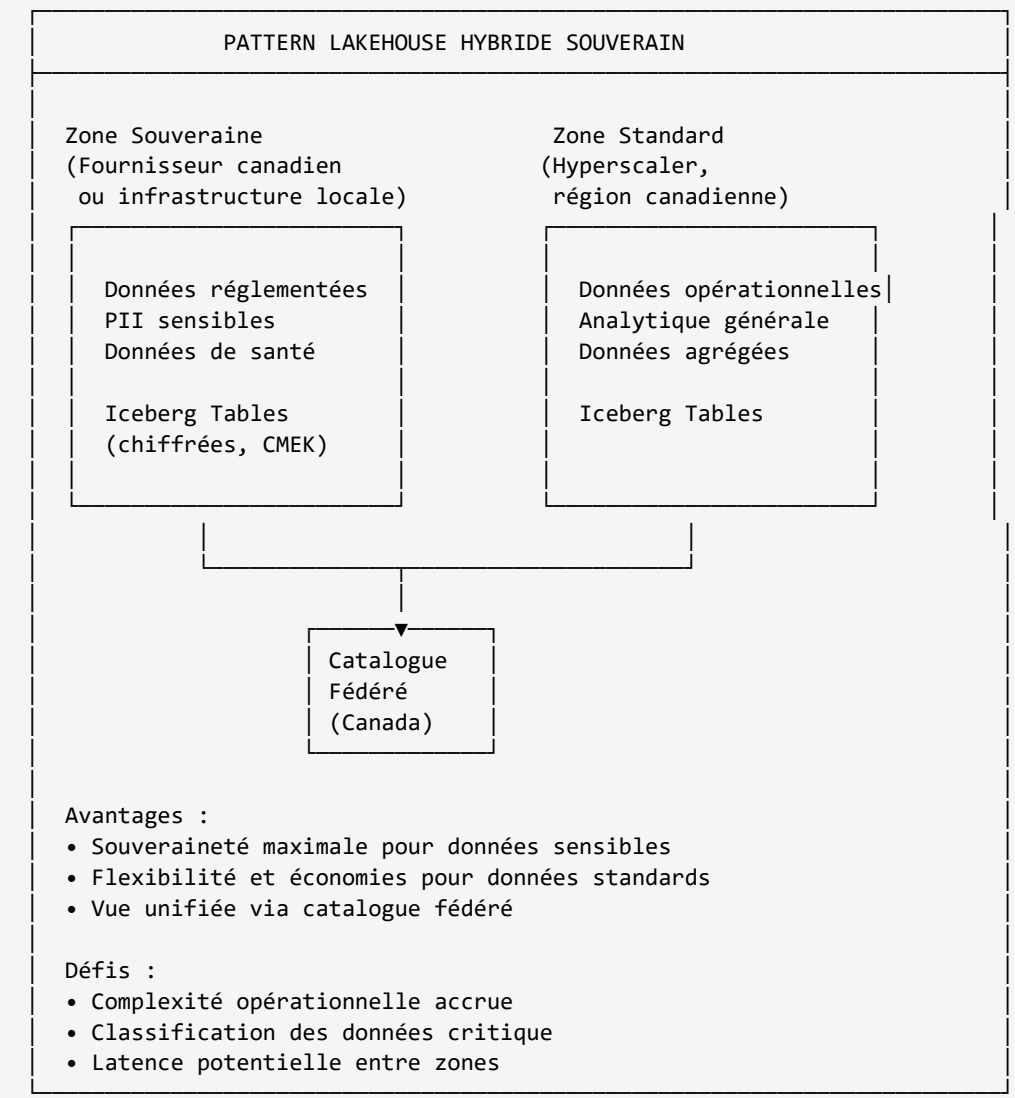
L'architecture événementielle comme fondation, à l'image de RBC avec Confluent Platform, permet de supporter à la fois les cas d'utilisation temps réel et analytique tout en maintenant une source unique de vérité. Le Volume III de cette monographie détaille les patterns d'intégration Kafka-Iceberg.

La gouvernance unifiée est essentielle dans un environnement réglementaire complexe. Un catalogue centralisé avec des politiques cohérentes facilite la conformité et réduit le risque d'incohérences entre les environnements.

Patterns d'Architecture Recommandés

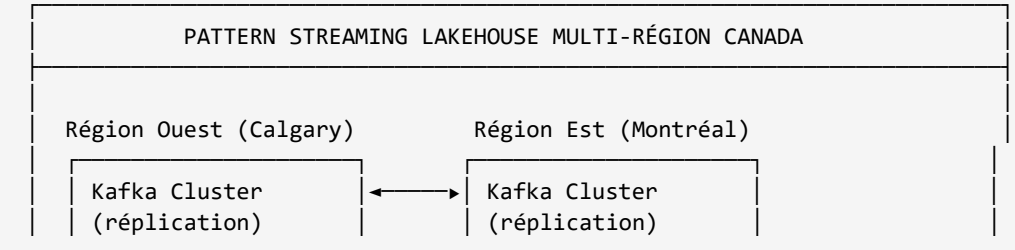
Pattern 1 : Lakehouse Hybride Souverain

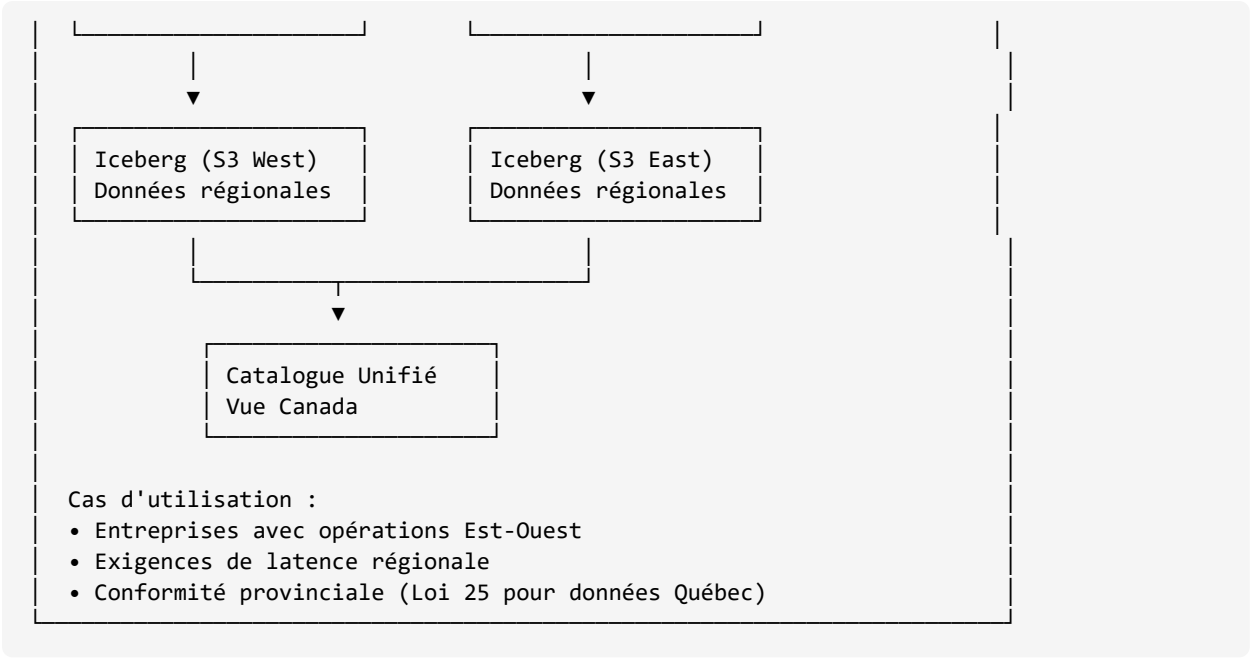
Pour les organisations avec des exigences de souveraineté strictes sur certaines données mais un besoin de flexibilité pour d'autres :



Pattern 2 : Streaming Lakehouse Multi-Région

Pour les organisations opérant à travers le Canada avec des exigences de latence régionale :




Iceberg (S3 West)
Données régionales
Iceberg (S3 East)
Données régionales
Catalogue Unifié
Vue Canada
Cas d'utilisation :
• Entreprises avec opérations Est-Ouest
• Exigences de latence régionale
• Conformité provinciale (Loi 25 pour données Québec)

Checklist de Conformité Lakehouse

Domaine	Vérification	Iceberg Pertinent
LPRPDE	Mécanismes de consentement documentés	Time travel pour audit
LPRPDE	Processus de demande d'accès établi	Requêtes ad hoc sur historique
LPRPDE	Mesures de sécurité appropriées	Chiffrement, contrôles d'accès
Loi 25	EFVP complétée pour le projet	Tables de métadonnées
Loi 25	Responsable protection renseignements désigné	Contrôles d'accès nominatifs
Loi 25	Droit à la portabilité supporté	Export Parquet/CSV natif
Loi 25	Évaluations transfrontalières documentées	Partitionnement par région
BSIF (si applicable)	Résilience opérationnelle testée	Récupération après sinistre
BSIF (si applicable)	Gestion des tiers documentée	Catalogue centralisé
BSIF (si applicable)	Tests de scénarios complétés	Restauration via snapshots
Souveraineté	Classification des données complète	Partitionnement par sensibilité
Souveraineté	Stockage dans régions canadiennes	Configuration buckets régionaux
Souveraineté	Politique de chiffrement établie	Chiffrement au repos/en transit
Souveraineté	Gestion des clés documentée	CMEK ou équivalent

Défis de Mise en Œuvre et Solutions

Les architectes implémentant des lakehouse au Canada rencontrent des défis communs qui méritent une attention particulière :

Défi 1 : Disponibilité des compétences

Le marché canadien des talents en data engineering est compétitif, particulièrement pour les compétences spécialisées en Apache Iceberg, Kafka et architectures lakehouse. Les organisations doivent :

- Investir dans la formation continue des équipes existantes
- Établir des partenariats avec des firmes de conseil spécialisées
- Considérer des modèles hybrides avec des équipes offshore supervisées localement

Défi 2 : Complexité réglementaire interprovinciale

Opérer à travers plusieurs provinces canadiennes implique de naviguer des exigences parfois contradictoires. La Loi 25 du Québec est plus restrictive que les lois des autres provinces, créant des défis pour les architectures pancandiennes. Solutions :

- Concevoir pour le dénominateur commun le plus strict

- Implémenter une segmentation des données par juridiction
- Automatiser les contrôles de conformité dans les pipelines

Défi 3 : Intégration avec les systèmes existants

La majorité des organisations canadiennes opèrent des environnements hétérogènes avec des systèmes legacy significatifs. L'adoption du lakehouse doit coexister avec :

- Les data warehouses existants (souvent Teradata, Oracle, ou solutions mainframe)
- Les systèmes de reporting établis
- Les processus ETL batch traditionnels

> **Migration**
> *De* : Architecture data traditionnelle (warehouse + data lake)
> *Vers* : Data Lakehouse unifié avec gouvernance canadienne
> *Stratégie* :
> 1. Établir le lakehouse en parallèle sans perturber les systèmes existants
> 2. Migrer les cas d'utilisation nouveaux en priorité
> 3. Migrer progressivement les charges de travail existantes par domaine
> 4. Maintenir une période de coexistence avec synchronisation bidirectionnelle
> 5. Décommissionner les systèmes legacy une fois la migration validée

Évolution du Paysage Canadien

Le contexte canadien des données continue d'évoluer rapidement. Plusieurs développements méritent une attention continue de la part des architectes :

Modernisation Législative Anticipée

La modernisation de la LPRPDE reste à l'ordre du jour fédéral. Le gouvernement du Canada a tenté à plusieurs reprises de moderniser la loi fédérale sur la protection des renseignements personnels, avec des propositions s'inspirant du RGPD européen. Le projet de loi C-27 (Loi sur la mise en œuvre de la Charte numérique) proposait notamment :

- Un nouveau Tribunal de la protection des renseignements personnels et des données
- Des pénalités administratives plus sévères
- Des exigences de transparence algorithmique pour les systèmes de décision automatisée
- Un droit à la mobilité des données comparable à celui de la Loi 25

Bien que le projet de loi n'ait pas été adopté dans sa forme originale, les architectes devraient concevoir leurs systèmes pour accommoder des exigences potentiellement plus strictes. La convergence vers un modèle de type RGPD semble inévitable à moyen terme.

Expansion de l'Infrastructure Infonuagique

L'expansion de l'infrastructure infonuagique canadienne s'accélère. Les annonces récentes incluent :

- Les Local Zones AWS à Toronto et Vancouver permettront des déploiements à faible latence dans ces métropoles
- L'expansion des centres de données Bell en Colombie-Britannique avec 500 MW de capacité
- La disponibilité d'Oracle Database@Google Cloud au Canada depuis décembre 2025, incluant le support d'Apache Iceberg via Oracle Autonomous AI Lakehouse

- L'investissement continu des opérateurs de centres de données comme Cologix et eStruxture

Cette expansion améliore les options de déploiement tout en maintenant la résidence des données au Canada. Pour les architectes lakehouse, cela signifie plus de choix pour le déploiement de clusters de traitement, de solutions de stockage et de services gérés.

Émergence des Cadres de Gouvernance de l'IA

Alors que les organisations canadiennes accélèrent leurs initiatives d'IA, de nouveaux cadres réglementaires et standards de l'industrie se développent. La Directive du Conseil du Trésor sur la prise de décisions automatisée impose déjà des exigences aux ministères fédéraux utilisant l'IA pour des décisions affectant les droits ou intérêts des personnes.

Pour les lakehouse servant de fondation aux initiatives d'IA, ces développements impliquent :

- Des exigences de traçabilité des données utilisées pour l'entraînement des modèles
- Des mécanismes d'explicabilité des décisions algorithmiques
- Des audits de biais et d'équité nécessitant l'accès aux données historiques
- Des processus de gouvernance documentés pour le cycle de vie des modèles

Apache Iceberg, avec ses capacités de time travel et de gestion des métadonnées, offre une fondation technique solide pour satisfaire ces exigences émergentes.

Tendances Sectorielles

Plusieurs tendances sectorielles canadiennes influencent l'adoption du lakehouse :

Dans le secteur financier, l'initiative DCM du BSIF pousse les institutions vers des architectures de données plus modernes et granulaires. Les exigences de déclaration réglementaire en temps quasi réel favorisent les approches streaming lakehouse.

Dans le secteur des télécommunications, la densification des réseaux 5G et l'expansion de l'IoT génèrent des volumes de données sans précédent. Les télécommunicateurs canadiens investissent massivement dans les capacités analytiques pour monétiser ces données.

Dans le secteur de la santé, l'intégration des données entre les systèmes provinciaux et la poussée vers les dossiers de santé électroniques interopérables créent des opportunités pour des plateformes de données unifiées.

Dans le secteur public, les initiatives de gouvernement numérique et d'ouverture des données encouragent l'adoption d'architectures modernes tout en maintenant des exigences strictes de souveraineté.

Feuille de Route pour l'Adoption

Pour les organisations canadiennes envisageant l'adoption d'une architecture Data Lakehouse, la feuille de route suivante offre un cadre structuré :

Phase 1 : Évaluation et Conception (3-6 mois) - Inventaire des données et classification par sensibilité - Analyse des exigences réglementaires applicables - Évaluation des options d'infrastructure (souveraine vs. hyperscaler) - Conception de l'architecture cible avec patterns appropriés - Définition de la stratégie de gouvernance

Phase 2 : Fondation (6-12 mois) - Déploiement de l'infrastructure de base (stockage, catalogue) - Établissement des pipelines d'ingestion initiaux - Mise en place des contrôles de sécurité et d'accès - Migration d'un domaine de données pilote - Validation de la conformité réglementaire

Phase 3 : Expansion (12-24 mois) - Migration progressive des domaines de données additionnels - Intégration des cas d'utilisation analytiques - Déploiement des capacités d'IA/ML - Optimisation des performances et des coûts - Formation et développement des compétences

Phase 4 : Maturité (24+ mois) - Décommissionnement des systèmes legacy - Industrialisation des processus DataOps - Évolution vers le streaming lakehouse - Optimisation continue et innovation

Résumé

Ce chapitre a exploré le contexte canadien unique dans lequel les architectes data conçoivent et déploient des Data Lakehouse. Les principaux enseignements peuvent être synthétisés comme suit :

Cadre Réglementaire Complexe et Évolutif

Le Canada présente un paysage réglementaire complexe combinant des exigences fédérales (LPRPDE avec ses dix principes fondamentaux) et provinciales (Loi 25 au Québec) avec des réglementations sectorielles (BSIF pour les institutions financières). La Loi 25 représente l'une des législations de protection des données les plus strictes en Amérique du Nord, imposant des évaluations des facteurs relatifs à la vie privée, le droit à la portabilité des données (effectif depuis septembre 2024), des exigences de consentement explicite et des pénalités pouvant atteindre 25 millions CAD ou 4 % du chiffre d'affaires mondial. Ces cadres doivent être intégrés dès la conception de l'architecture lakehouse, avec Apache Iceberg offrant des capacités natives de time travel et de gestion des métadonnées qui facilitent la conformité.

Étude de Cas RBC : Excellence en IA et Architecture Événementielle

La Banque Royale du Canada illustre une approche exemplaire de modernisation des données à grande échelle dans le secteur financier. Son architecture événementielle basée sur Confluent Platform, son nuage privé IA développé avec Red Hat et NVIDIA via Borealis AI (950+ employés), et son classement #3 mondial pour la maturité IA (Evident AI Index 2024) démontrent qu'une institution financière canadienne peut atteindre l'excellence technologique tout en respectant les contraintes réglementaires strictes du BSIF. L'analyse de 11 billions d'événements de sécurité en 2024 et la plateforme de trading IA Aiden témoignent de l'échelle et de la sophistication atteintes. L'architecture de référence inspirée de RBC combine backbone événementiel, lakehouse Iceberg souverain et infrastructure IA dédiée.

Étude de Cas Bell Canada : Transformation Télécommunications et Souveraineté

Bell Canada représente la transformation numérique d'un grand télécommunicateur canadien avec 22 millions de connexions clients. Son partenariat stratégique avec Google Cloud pour la solution Network AI Ops (amélioration significative du MTTR), son alliance avec ServiceNow faisant de Bell l'un des plus grands clients communications de la plateforme, et son développement de six centres de données IA en Colombie-Britannique (500 MW de capacité) avec une cible de revenus IA de 1,07 milliard CAD d'ici 2028 illustrent l'ambition technologique. Le lancement de Bell Cyber répond directement aux préoccupations de souveraineté des données : selon l'étude Bell Enterprise AI (octobre 2025), 90 % des dirigeants affirment qu'il est plus important que jamais de conserver les données sensibles au Canada.

Distinction Fondamentale : Résidence versus Souveraineté

La distinction entre résidence des données (où les données sont physiquement stockées) et souveraineté des données (qui a l'autorité légale) est fondamentale mais souvent mal comprise. La CLOUD Act américaine de 2018 permet aux autorités américaines d'accéder aux données contrôlées par des entreprises américaines (AWS, Azure, GCP), même si ces données résident au Canada. Le gouvernement du Canada reconnaît dans son livre blanc que tant qu'un fournisseur opérant au Canada est assujetti aux lois

étrangères, le Canada n'aura pas la souveraineté complète. Les architectes doivent évaluer leur profil de risque et choisir des stratégies appropriées : fournisseurs canadiens pour la souveraineté maximale, clauses contractuelles robustes, chiffrement avec gestion des clés par le client (CMEK), ou architecture hybride segmentant les données par sensibilité.

Infrastructure Régionale en Expansion

Le Canada dispose maintenant d'une infrastructure infonuagique substantielle : AWS (Montréal avec 3 ZD depuis 2016, Calgary avec 3 ZD depuis 2023, Local Zones prévues à Toronto et Vancouver), Azure (Toronto et Québec depuis 2016), Google Cloud (Montréal et Toronto). Cette infrastructure permet la résidence des données au Canada tout en bénéficiant des services infonuagiques modernes. L'écosystème s'enrichit avec des opérateurs canadiens (Cologix, eStruxture) et l'expansion de Bell dans les centres de données IA. Apache Iceberg, avec son architecture ouverte et son agnosticisme au stockage, est particulièrement bien adapté aux environnements multi-cloud et hybrides courants au Canada.

Considérations Économiques et Défis de Mise en Œuvre

Les régions canadiennes présentent un premium de coûts de 5-15 % par rapport aux régions américaines, mais ce premium doit être mis en balance avec les coûts de conformité évités et les risques juridiques. Les défis de mise en œuvre incluent la disponibilité des compétences spécialisées, la complexité réglementaire interprovinciale et l'intégration avec les systèmes legacy. Les patterns d'architecture recommandés (Lakehouse Hybride Souverain, Streaming Lakehouse Multi-Région) offrent des approches structurées pour naviguer ces défis.

Recommandations Clés pour les Architectes Canadiens

Les architectes canadiens devraient adopter une approche de conformité dès la conception, comprendre la distinction résidence/souveraineté, favoriser les formats ouverts (Iceberg, Parquet) pour la portabilité, considérer l'architecture événementielle (Kafka) comme fondation, et implémenter une gouvernance unifiée via un catalogue centralisé. La feuille de route en quatre phases (Évaluation, Fondation, Expansion, Maturité) offre un cadre structuré pour l'adoption. Le Data Lakehouse basé sur Apache Iceberg offre la flexibilité nécessaire pour naviguer le contexte réglementaire et infrastructurel canadien unique tout en supportant les initiatives d'IA et d'analytique avancée qui sont au cœur de la transformation numérique des organisations canadiennes.

Ressources et Références Clés pour les Architectes Canadiens

Pour approfondir les sujets traités dans ce chapitre, les architectes canadiens peuvent consulter les ressources suivantes :

Domaine	Ressource	Organisation
LPRPDE	Guide sur la protection des renseignements personnels	Commissariat à la protection de la vie privée du Canada
Loi 25	Lignes directrices sur la protection des renseignements personnels	Commission d'accès à l'information du Québec (CAI)
BSIF	Ligne directrice B-13 : Gestion du risque technologique et cybernétique	Bureau du surintendant des institutions financières
BSIF	Initiative DCM (Modernisation de la collecte de données)	BSIF, Banque du Canada, SADC
Souveraineté	Livre blanc sur la souveraineté des données et l'infonuagique publique	Gouvernement du Canada
Infonuagique	Stratégie d'adoption de l'infonuagique du GC	Secrétariat du Conseil du Trésor

Les communautés professionnelles canadiennes offrent également des opportunités de partage d'expériences :

- La communauté Apache Kafka Canada regroupe les praticiens du streaming de données
- Les groupes d'utilisateurs Spark et Iceberg dans les principales villes canadiennes
- Les associations professionnelles comme ISACA Canada et l'AQIII au Québec
- Les conférences comme Data Science GO (Toronto) et MTL Connect (Montréal)

Le chapitre suivant conclut ce volume avec une perspective sur l'évolution du Data Lakehouse à l'horizon 2026-2030, explorant les tendances technologiques émergentes et leurs implications stratégiques pour les organisations canadiennes.

Chapitre IV.16 - Conclusion Finale et Perspectives 2026-2030

Introduction

Au terme de ce parcours approfondi dans l'univers d'Apache Iceberg et du Data Lakehouse moderne, nous arrivons à un moment charnière pour l'architecture des données d'entreprise. Les quinze chapitres précédents ont établi les fondations conceptuelles, techniques et opérationnelles nécessaires à la conception et à l'exploitation d'un lakehouse robuste. Ce chapitre de conclusion synthétise les apprentissages clés, analyse l'état actuel de l'écosystème et projette les évolutions majeures qui façonneront le paysage des données entre 2026 et 2030.

L'adoption d'Apache Iceberg s'est accélérée de manière remarquable depuis 2023. Ce qui était alors une technologie prometteuse principalement utilisée par les géants technologiques est devenu, en 2025, le standard de facto pour les architectures lakehouse en entreprise. Cette trajectoire n'est pas le fruit du hasard : elle résulte de la convergence de facteurs technologiques, économiques et organisationnels qui ont rendu le modèle lakehouse non seulement viable, mais souvent supérieur aux approches traditionnelles.

Pour les architectes données et les ingénieurs de données qui ont suivi ce volume, la question n'est plus de savoir si Iceberg mérite considération, mais plutôt comment maximiser la valeur de son adoption dans un contexte d'évolution technologique rapide. Les cinq prochaines années apporteront des transformations significatives : l'émergence de l'intelligence artificielle générative comme consommatrice majeure de données, la standardisation des interfaces de catalogue, l'optimisation automatisée par l'apprentissage machine et l'intégration native avec les architectures agentiques.

Ce chapitre final offre une perspective stratégique pour naviguer ces transformations. Nous commencerons par une synthèse des principes fondamentaux établis dans ce volume, avant d'examiner l'état de l'écosystème en 2025-2026. Nous explorerons ensuite les tendances technologiques majeures anticipées pour 2026-2030, les dynamiques de marché et de consolidation, puis nous formulerons des recommandations stratégiques concrètes. Nous conclurons par une feuille de route d'adoption adaptée aux différents contextes organisationnels et une vision pour l'avenir du lakehouse d'entreprise.

Synthèse des Fondamentaux Iceberg

Les Piliers Architecturaux

Au fil de ce volume, nous avons établi qu'Apache Iceberg repose sur quatre piliers architecturaux fondamentaux qui le distinguent des formats de table traditionnels et des approches data lake de première génération.

Le premier pilier, la **séparation du stockage et du calcul** , représente bien plus qu'une simple commodité infonuagique. Cette séparation permet une élasticité indépendante des ressources de stockage et de traitement, une optimisation des coûts par niveau de stockage (chaud, tiède, froid), une scalabilité horizontale sans limite théorique et une résilience accrue par l'isolation des composantes. Les organisations qui ont adopté cette approche rapportent typiquement des réductions de coûts de 40 à 60 % comparativement aux architectures data warehouse traditionnelles, tout en gagnant en flexibilité opérationnelle.

Le deuxième pilier concerne la **couche de métadonnées sophistiquée** . Contrairement aux formats qui stockent les métadonnées dans des fichiers manifestes simples ou qui dépendent entièrement d'un metastore externe, Iceberg maintient une hiérarchie de métadonnées complète : fichiers de métadonnées pointant vers des listes de manifestes, elles-mêmes référençant des fichiers manifestes qui décrivent les fichiers de données. Cette architecture permet le suivi précis de chaque modification (snapshots), la navigation temporelle (time travel), la planification de requêtes efficace sans scanner l'ensemble des données et la gestion des conflits de concurrence optimiste.

Le troisième pilier est l' **évolution de schéma sécurisée** . Dans un environnement où les besoins d'affaires évoluent constamment, la capacité de modifier le schéma d'une table sans réécrire les données existantes constitue un avantage considérable. Iceberg supporte l'ajout, la suppression, le renommage et la réorganisation des colonnes, ainsi que la modification des types de données (avec des règles de promotion sécurisées). Les données historiques restent accessibles avec leur schéma d'origine, tandis que les nouvelles requêtes bénéficient du schéma actualisé.

Le quatrième pilier, le **partitionnement masqué** , élimine une source majeure d'erreurs et de complexité dans les architectures data lake traditionnelles. Les utilisateurs n'ont plus besoin de connaître le schéma de partitionnement pour écrire des requêtes efficaces : Iceberg applique automatiquement le filtrage de partitions basé sur les prédicats de la requête. De plus, l'évolution du partitionnement permet d'adapter la stratégie au fil du temps sans migration de données.

Leçons Opérationnelles

Les chapitres consacrés aux opérations ont mis en lumière plusieurs leçons essentielles pour le succès d'un déploiement Iceberg en production.

La **maintenance proactive** s'avère indispensable. Un lakehouse Iceberg non maintenu accumule des fichiers de petite taille, des snapshots obsolètes et des métadonnées fragmentées, dégradant progressivement les performances. Les organisations matures établissent des processus automatisés de compaction, d'expiration des snapshots et de nettoyage des fichiers orphelins. La fréquence de ces opérations dépend du volume d'ingestion et des patrons d'accès, mais un cycle quotidien ou hebdomadaire constitue généralement un bon point de départ.

La **gouvernance intégrée** dès la conception représente un facteur de succès critique. Plutôt que d'ajouter des contrôles de sécurité et de conformité après coup, les architectures lakehouse performantes intègrent ces considérations dès les premières phases de conception. Cela inclut la définition des politiques d'accès au niveau des tables et des colonnes, le catalogage et le lignage des données, la classification des données sensibles et les mécanismes d'audit.

L'**observabilité complète** permet d'anticiper les problèmes avant qu'ils n'affectent les utilisateurs. Les métriques clés à surveiller incluent la taille et le nombre de fichiers par table, le temps de planification des requêtes, les taux de succès et d'échec des opérations de maintenance, l'utilisation du stockage par niveau et l'évolution des coûts. Les tableaux de bord opérationnels doivent offrir une visibilité en temps réel sur ces indicateurs.

Architecture de Référence Consolidée

En consolidant les apprentissages des chapitres précédents, l'architecture de référence d'un lakehouse Iceberg d'entreprise comprend les couches suivantes :

La **couche de stockage** utilise un stockage objet compatible S3 (Amazon S3, Azure Data Lake Storage Gen2, Google Cloud Storage ou MinIO pour les déploiements sur site). Les données sont organisées par domaine métier avec une stratégie de stockage par niveaux basée sur la fréquence d'accès.

La **couche de catalogue** centralise les métadonnées dans un catalogue REST conforme à la spécification Iceberg. Les options principales incluent Nessie pour les capacités de branchement Git-like, le catalogue REST de Dremio pour l'intégration avec leur moteur de requête, ou les catalogues gérés des fournisseurs infonuagiques.

La **couche d'ingestion** combine des pipelines par lots (Apache Spark, Trino, Dremio) avec des flux en continu (Apache Kafka avec Kafka Connect ou Apache Flink). L'intégration avec les patterns CDC (Change Data Capture) permet la synchronisation des données transactionnelles.

La **couche de traitement** offre une fédération de moteurs adaptés aux différents cas d'usage : Trino pour les requêtes interactives ad hoc, Apache Spark pour les transformations massives et l'apprentissage machine, Dremio pour la sémantique et l'accélération de requêtes.

La **couche de consommation** expose les données via des interfaces adaptées aux différents consommateurs : connecteurs BI (Power BI, Tableau), interfaces SQL (JDBC/ODBC), APIs REST pour les applications et accès programmatique pour les pipelines d'apprentissage machine.

État de l'Écosystème 2025-2026

Consolidation du Marché

L'écosystème Iceberg a connu une maturation significative entre 2023 et 2026. Plusieurs tendances de consolidation caractérisent l'état actuel du marché.

L'**adoption par les grands fournisseurs** a atteint un niveau sans précédent. Amazon Web Services offre une intégration native d'Iceberg dans Amazon Athena, Amazon EMR, AWS Glue et Amazon Redshift Spectrum. Microsoft a intégré Iceberg dans Fabric via les raccourcis OneLake, permettant l'accès aux tables Iceberg depuis Power BI Direct Lake. Google Cloud propose le support Iceberg dans BigQuery et Dataproc. Snowflake a ajouté le support des tables externes Iceberg. Cette convergence des grands acteurs valide Iceberg comme standard de l'industrie et réduit le risque d'adoption pour les entreprises.

La **spécialisation des acteurs** s'affirme également. Dremio se positionne comme la plateforme lakehouse unifiée avec son moteur de requête optimisé et son catalogue REST. Tabular (fondé par les créateurs d'Iceberg) offre des services de gestion de catalogue et d'optimisation automatique. Starburst consolide sa position sur Trino avec des fonctionnalités entreprise. Confluent intègre Iceberg dans sa stratégie de streaming lakehouse avec Tableflow. Cette spécialisation favorise l'innovation tout en offrant aux entreprises des options claires selon leurs priorités.

L'**interopérabilité comme norme** s'est imposée. Le protocole REST Catalog standardise les interactions avec les métadonnées Iceberg, permettant aux clients de différents moteurs d'accéder aux mêmes tables. Les formats de fichiers (Parquet, ORC, Avro) restent ouverts et lisibles par une multitude d'outils. Cette

ouverture contraste avec les approches propriétaires et constitue un argument majeur pour l'adoption d'Iceberg.

Maturité Technologique

Sur le plan technologique, plusieurs fonctionnalités ont atteint la maturité en production.

L'**évolution de schéma** fonctionne de manière fiable dans tous les scénarios courants. Les organisations utilisent cette capacité pour adapter leurs modèles de données aux évolutions des besoins d'affaires sans interruption de service ni migration complexe.

Le **partitionnement masqué** a prouvé sa valeur pour simplifier les requêtes et optimiser les performances. L'évolution du partitionnement permet désormais de modifier la stratégie de partitionnement sans réécrire les données historiques, une fonctionnalité particulièrement utile lorsque les patrons d'accès changent.

Le **time travel** est largement utilisé pour l'audit, la reproduction des résultats analytiques et la récupération après erreur. Les organisations établissent des politiques de rétention adaptées à leurs besoins de conformité et de reprise après sinistre.

Les **opérations de maintenance** (compaction, expiration des snapshots, nettoyage des orphelins) sont automatisées dans la plupart des déploiements matures. Les outils de gestion comme ceux de Tabular ou les fonctionnalités intégrées de Dremio simplifient ces opérations.

Défis Persistants

Malgré cette maturité, certains défis persistent dans l'écosystème Iceberg.

La **gestion des petits fichiers** reste un défi opérationnel. Les charges de travail en streaming ou les mises à jour fréquentes génèrent naturellement de nombreux petits fichiers qui dégradent les performances. Bien que la compaction résolve ce problème, elle nécessite une planification et des ressources dédiées.

La **complexité des métadonnées** peut devenir problématique pour les tables très volumineuses avec un historique long. La planification des requêtes doit traverser la hiérarchie de métadonnées, ce qui peut introduire une latence perceptible. Les optimisations comme le filtrage de partitions au niveau des manifestes et la mise en cache des métadonnées atténuent ce problème.

L'**expertise requise** pour concevoir et opérer un lakehouse Iceberg reste significative. Les équipes doivent maîtriser les concepts de stockage objet, les moteurs de traitement distribué, la gouvernance des données et les opérations de maintenance. Cette courbe d'apprentissage peut freiner l'adoption dans les organisations avec des ressources limitées.

La **standardisation du catalogue** progresse mais n'est pas encore complète. Bien que le protocole REST Catalog soit largement adopté, des variations d'implémentation subsistent entre les fournisseurs. Les fonctionnalités avancées comme le branchement (Nessie) ou les politiques d'accès ne sont pas standardisées.

Tendances Technologiques 2026-2030

Intelligence Artificielle et Lakehouse

L'intégration entre l'intelligence artificielle et le lakehouse représente la tendance la plus transformatrice pour les cinq prochaines années.

L'**IA générative comme consommatrice de données** modifie fondamentalement les patrons d'accès au lakehouse. Les modèles de langage et les systèmes RAG (Retrieval-Augmented Generation) nécessitent un accès efficace à de vastes corpus de données structurées et non structurées. Le lakehouse Iceberg, avec sa capacité à gérer des données de différents formats et sa scalabilité, devient la couche de persistance privilégiée pour ces charges de travail. Les organisations avant-gardistes conçoivent déjà leurs lakehouses avec les cas d'usage d'IA en tête, incluant le stockage des embeddings vectoriels, le versionnement des jeux de données d'entraînement et la traçabilité des données utilisées par les modèles.

L'**optimisation automatisée par l'apprentissage machine** transformera les opérations lakehouse. Les systèmes apprendront des patrons d'accès historiques pour optimiser automatiquement le partitionnement, suggérer les index et les vues matérialisées appropriés, planifier les opérations de compaction au moment optimal et prédire les besoins en ressources. Cette auto-optimisation réduira la charge opérationnelle et améliorera les performances sans intervention humaine.

Les **interfaces en langage naturel** démocratiseront l'accès aux données du lakehouse. Les utilisateurs d'affaires pourront interroger les données en formulant des questions en français ou en anglais, le système traduisant automatiquement en requêtes SQL optimisées. Cette capacité, déjà émergente en 2025, deviendra mature et fiable d'ici 2028, réduisant la dépendance envers les analystes techniques pour les requêtes ad hoc.

Évolution du Format Iceberg

Le format Iceberg lui-même continuera d'évoluer pour répondre aux besoins émergents.

Les **améliorations de performance** incluront des optimisations au niveau des métadonnées pour les tables très volumineuses, des formats de fichiers plus efficaces pour les charges de travail analytiques et une meilleure gestion de la concurrence pour les scénarios à haute fréquence de mise à jour.

Le **support des types de données avancés** s'étendra pour inclure les types géospatiaux natifs, les embeddings vectoriels pour l'IA, les types de données de séries temporelles optimisés et les structures de données imbriquées plus complexes. Ces extensions permettront de stocker et d'interroger efficacement des données qui nécessitent aujourd'hui des solutions spécialisées.

L'**intégration native du streaming** évoluera au-delà des connecteurs actuels. Les futures versions d'Iceberg pourraient inclure des primitives pour le traitement en temps réel, réduisant la latence entre l'ingestion et la disponibilité pour les requêtes. Le concept de « streaming lakehouse » décrit dans les chapitres précédents deviendra plus fluide et plus performant.

Standardisation de l'Écosystème

La standardisation jouera un rôle crucial dans la maturation de l'écosystème.

Le **protocole REST Catalog** deviendra le standard universel pour l'accès aux métadonnées Iceberg. Les variations actuelles entre implémentations convergeront vers une spécification commune, facilitant l'interopérabilité entre les outils de différents fournisseurs. Les fonctionnalités avancées comme le branchement et les politiques d'accès seront intégrées dans cette spécification.

L'**interopérabilité entre formats de table** progressera. Des initiatives comme Apache XTable (anciennement OneTable) permettent déjà de convertir entre Iceberg, Delta Lake et Hudi. D'ici 2030, cette interopérabilité sera plus transparente, permettant aux organisations de choisir leur format préféré tout en conservant la flexibilité de migrer ou de fédérer des données entre formats.

Les **standards de gouvernance** émergeront pour harmoniser les approches de sécurité, de lignage et de qualité des données dans les environnements lakehouse multi-moteurs. Ces standards faciliteront la conformité réglementaire et la collaboration entre organisations.

Architecture Agentique et Lakehouse

L'émergence des architectures agentiques, thème central de cette monographie, influencera profondément l'évolution du lakehouse.

Les **agents autonomes** consommeront et produiront des données dans le lakehouse. Ces agents, capables d'exécuter des tâches complexes de manière autonome, nécessitent un accès fiable et performant aux données historiques et en temps réel. Le lakehouse Iceberg, intégré avec le backbone événementiel Kafka décrit dans le Volume III, fournira cette infrastructure de données.

La **traçabilité des décisions** deviendra critique dans un contexte où des agents prennent des décisions autonomes. Le time travel d'Iceberg et les capacités de lignage permettront de reconstruire l'état des données au moment où une décision a été prise, une exigence pour l'audit et la conformité des systèmes agentiques.

L'**orchestration événementielle** entre le lakehouse et les agents se sophistiquera. Les changements de données dans le lakehouse déclencheront des actions d'agents, et les résultats des agents alimenteront le lakehouse. Cette boucle de rétroaction nécessite une intégration étroite entre Iceberg et les plateformes d'orchestration agentique.

Dynamiques de Marché et Consolidation

Évolution du Paysage Concurrentiel

Le marché du data lakehouse connaîtra des transformations significatives entre 2026 et 2030.

La **consolidation des acteurs** se poursuivra. Les startups spécialisées seront acquises par des acteurs majeurs cherchant à compléter leur offre. Les fournisseurs infonuagiques continueront d'intégrer les fonctionnalités lakehouse dans leurs plateformes, augmentant la pression concurrentielle sur les acteurs indépendants. Cette consolidation simplifiera le paysage pour les entreprises mais réduira potentiellement l'innovation de rupture.

La **différenciation par la valeur ajoutée** deviendra le principal axe de compétition. Puisque le format Iceberg est ouvert, les fournisseurs ne peuvent pas se différencier par le format lui-même. La compétition se concentrera sur les performances du moteur de requête, les fonctionnalités de gouvernance, l'automatisation des opérations, l'intégration avec les outils d'IA et la qualité du support.

L'**émergence de solutions verticales** répondra aux besoins spécifiques de certains secteurs. Des solutions lakehouse optimisées pour la finance (conformité, audit), la santé (données sensibles, réglementation), les télécommunications (volumes massifs, temps réel) et d'autres secteurs se développeront, offrant des fonctionnalités préconfigurées et des certifications de conformité.

Impact sur les Décisions d'Entreprise

Ces dynamiques de marché influenceront les stratégies d'adoption des entreprises.

La **réduction du risque de verrouillage** restera une priorité. L'adoption d'Iceberg comme format ouvert atténue le risque de verrouillage auprès d'un fournisseur unique. Les entreprises pourront migrer entre moteurs de requête et fournisseurs infonuagiques tout en conservant leurs données et métadonnées. Cette portabilité deviendra un critère de sélection majeur.

L'**équilibre entre gestion interne et services gérés** évoluera. Pour les grandes organisations avec des équipes data matures, la gestion interne d'un lakehouse Iceberg offre un contrôle maximal mais nécessite des investissements en compétences et en opérations. Les services gérés (Dremio Cloud, Tabular, catalogues des fournisseurs infonuagiques) réduisent la charge opérationnelle mais introduisent une dépendance. La tendance sera vers des modèles hybrides où les composantes critiques sont gérées en interne tandis que les fonctionnalités commoditisées sont externalisées.

Les **partenariats stratégiques** entre fournisseurs de lakehouse et fournisseurs de solutions d'entreprise (ERP, CRM, BI) se multiplieront. Ces partenariats faciliteront l'intégration des données d'entreprise dans le lakehouse et l'exposition des données lakehouse vers les applications métier.

Contexte Canadien

Le marché canadien présente des caractéristiques distinctives qui influencent l'adoption du lakehouse.

La **souveraineté des données** constitue une préoccupation majeure pour les organisations canadiennes, particulièrement dans les secteurs réglementés. La capacité de déployer un lakehouse Iceberg dans les régions infonuagiques canadiennes (Toronto, Montréal pour AWS et Azure) ou sur site répond à ces exigences. Les fournisseurs qui offrent des garanties de résidence des données au Canada bénéficient d'un avantage concurrentiel.

La **réglementation** canadienne, incluant la LPRPDE (Loi sur la protection des renseignements personnels et les documents électroniques) et les lois provinciales comme la Loi 25 au Québec, impose des obligations de gouvernance des données. Le lakehouse Iceberg, avec ses capacités de contrôle d'accès, de lignage et d'audit, facilite la conformité à ces réglementations.

L'**écosystème technologique** canadien, bien que plus petit que celui des États-Unis, inclut des acteurs significatifs dans le domaine des données. Les grandes banques canadiennes, les compagnies d'assurance, les opérateurs de télécommunications et les détaillants investissent dans les architectures lakehouse. Les fournisseurs de services professionnels (firmes de consultation, intégrateurs de systèmes) développent des pratiques spécialisées en lakehouse pour servir ce marché.

Recommandations Stratégiques

Pour les Organisations en Phase d'Évaluation

Les organisations qui n'ont pas encore adopté le lakehouse Iceberg devraient considérer les recommandations suivantes.

Commencer par un cas d'usage concret plutôt que par une transformation massive. Identifier un domaine de données avec des défis clairs (coûts élevés, performance insuffisante, flexibilité limitée) et

implémenter un lakehouse Iceberg pour ce domaine. Cette approche permet de développer l'expertise, de démontrer la valeur et d'apprendre des erreurs avant d'étendre l'adoption.

Évaluer les options de déploiement en fonction des ressources et des compétences disponibles. Les services gérés offrent un démarrage rapide avec moins de friction opérationnelle. La gestion interne offre plus de contrôle mais nécessite des investissements en compétences. Un modèle hybride peut combiner les avantages des deux approches.

Investir dans les compétences dès le début du projet. Former les ingénieurs de données aux concepts Iceberg, aux moteurs de requête et aux opérations de maintenance. Recruter ou développer une expertise en architecture lakehouse. Ces investissements porteront leurs fruits tout au long du parcours d'adoption.

Planifier la gouvernance avant de migrer des données. Définir les politiques d'accès, les standards de qualité, les exigences de conformité et les processus de catalogage. Intégrer ces considérations dans l'architecture plutôt que de les ajouter après coup.

Pour les Organisations en Phase de Mise à l'Échelle

Les organisations qui ont déjà adopté Iceberg et cherchent à étendre leur utilisation devraient considérer les recommandations suivantes.

Établir un centre d'excellence lakehouse qui consolide l'expertise, définit les standards et accompagne les équipes dans l'adoption. Ce centre d'excellence peut être une équipe dédiée ou une communauté de pratique distribuée, selon la structure organisationnelle.

Automatiser les opérations de maintenance, de surveillance et de réponse aux incidents. Les scripts manuels ne passent pas à l'échelle. Investir dans l'infrastructure as code, les pipelines CI/CD pour les artefacts de données et les outils d'observabilité intégrés.

Optimiser les coûts en analysant l'utilisation réelle du lakehouse. Identifier les tables sous-utilisées, les requêtes inefficaces et les opportunités de stockage par niveaux. Établir des mécanismes de refacturation qui responsabilisent les équipes consommatrices.

Préparer l'intégration avec l'IA en structurant les données pour les cas d'usage d'apprentissage machine et d'IA générative. Établir des processus pour le versionnement des jeux de données, la traçabilité des données d'entraînement et l'accès performant aux données pour l'inférence.

Pour les Organisations Matures

Les organisations avec un lakehouse Iceberg mature et éprouvé devraient considérer les recommandations suivantes.

Évaluer les capacités avancées comme le branchement de données (Nessie), l'intégration en temps réel avec Kafka et la fédération multi-lakehouse. Ces capacités peuvent débloquer de nouveaux cas d'usage et améliorer l'agilité des équipes de données.

Contribuer à l'écosystème en partageant les apprentissages, en participant aux communautés open source et en fournissant des retours aux fournisseurs. Cette contribution renforce l'écosystème dont l'organisation dépend et influence son évolution dans une direction favorable.

Planifier les évolutions architecturales pour les cinq prochaines années. Anticiper l'intégration avec les architectures agentiques, l'évolution des besoins en IA, les changements réglementaires et les innovations technologiques. Une feuille de route architecturale permet de prendre des décisions cohérentes et d'éviter les impasses techniques.

Mesurer et communiquer la valeur générée par le lakehouse. Établir des indicateurs de performance (réduction des coûts, amélioration de la productivité, nouveaux cas d'usage activés) et les communiquer à la direction. Cette démonstration de valeur sécurise les investissements continus et renforce le positionnement de l'équipe data.

Feuille de Route d'Adoption

Phase 1 : Fondations (0-6 mois)

La première phase établit les fondations techniques et organisationnelles du lakehouse.

Objectifs :

- Déployer l'infrastructure de base (stockage, catalogue, moteur de requête)
- Migrer un premier cas d'usage pilote
- Former l'équipe noyau aux compétences essentielles
- Établir les processus opérationnels de base

Livrables :

- Architecture de référence documentée
- Environnement de développement et de production opérationnel
- Premier domaine de données migré vers Iceberg
- Documentation des procédures opérationnelles

Indicateurs de succès :

- Disponibilité du lakehouse supérieure à 99 %
- Performance des requêtes équivalente ou supérieure à l'ancien système
- Équipe capable d'opérer le lakehouse de manière autonome

Risques et mitigations :

- *Risque* : Sous-estimation de la complexité technique. *Mitigation* : Commencer par un cas d'usage simple et bien défini.
- *Risque* : Résistance au changement des équipes. *Mitigation* : Impliquer les utilisateurs clés dès le début et démontrer rapidement la valeur.

Phase 2 : Expansion (6-18 mois)

La deuxième phase étend l'adoption au-delà du pilote initial.

Objectifs :

- Migrer les domaines de données prioritaires
- Établir les standards de gouvernance et de qualité
- Automatiser les opérations de maintenance
- Développer les compétences à travers l'organisation

Livrables :

- Plusieurs domaines de données opérationnels dans le lakehouse
- Catalogue de données avec métadonnées et lignage

- Pipelines de maintenance automatisés
- Programme de formation pour les ingénieurs de données et les analystes

Indicateurs de succès :

- Nombre de domaines migrés selon le plan
- Adoption mesurée par le nombre d'utilisateurs et de requêtes
- Réduction des coûts de données par rapport à l'architecture précédente
- Satisfaction des utilisateurs mesurée par des enquêtes

Risques et mitigations :

- *Risque* : Accumulation de dette technique. *Mitigation* : Établir des revues d'architecture régulières et des critères de qualité.
- *Risque* : Croissance incontrôlée des coûts. *Mitigation* : Implémenter la surveillance des coûts et les mécanismes de refacturation.

Phase 3 : Optimisation (18-36 mois)

La troisième phase optimise le lakehouse pour la performance, les coûts et l'agilité.

Objectifs :

- Optimiser les performances des requêtes critiques
- Implémenter le stockage par niveaux pour réduire les coûts
- Intégrer les cas d'usage d'IA et d'apprentissage machine
- Établir l'observabilité avancée et l'auto-réparation

Livrables :

- Benchmarks de performance documentés et optimisés
- Stratégie de stockage par niveaux implémentée
- Infrastructure de données pour l'IA opérationnelle
- Tableau de bord d'observabilité complet

Indicateurs de succès :

- Amélioration mesurable des temps de réponse des requêtes
- Réduction des coûts de stockage grâce au stockage par niveaux
- Cas d'usage d'IA alimentés par le lakehouse
- Temps moyen de résolution des incidents réduit

Risques et mitigations :

- *Risque* : Optimisation prématurée. *Mitigation* : Prioriser les optimisations en fonction de l'impact mesuré.
- *Risque* : Complexité croissante. *Mitigation* : Maintenir une documentation à jour et des processus de revue.

Phase 4 : Innovation (36+ mois)

La quatrième phase positionne le lakehouse comme plateforme d'innovation.

Objectifs :

- Intégrer le lakehouse avec les architectures agentiques
- Implémenter les capacités avancées (branchement, temps réel, fédération)

- Établir le lakehouse comme plateforme de données d'entreprise
- Contribuer à l'écosystème et influencer son évolution

Livrables :

- Intégration avec les plateformes agentiques
- Capacités avancées opérationnelles
- Stratégie de plateforme de données documentée
- Contributions à l'écosystème open source

Indicateurs de succès :

- Nouveaux cas d'usage activés par les capacités avancées
- Reconnaissance comme leader en architecture données
- Influence sur l'évolution de l'écosystème

Risques et mitigations :

- *Risque* : Technologie qui évolue plus vite que l'adoption. *Mitigation* : Veille technologique continue et expérimentation contrôlée.
- *Risque* : Perte de focus sur les fondamentaux. *Mitigation* : Maintenir les indicateurs de base et prioriser la stabilité.

Études de Cas Prospectives

Cas 1 : Institution Financière Canadienne

> **Étude de cas : Banque Nationale du Canada (projection 2028)** *Secteur* : Services financiers *Contexte* : Grande banque canadienne avec des données distribuées entre plusieurs systèmes patrimoniaux et initiatives infonuagiques *Vision* : Lakehouse Iceberg comme couche de données unifiée pour l'analytique, la conformité et l'IA

Architecture projetée :

L'institution déploie un lakehouse Iceberg sur Azure Data Lake Storage Gen2 dans la région canadienne. Le catalogue REST, hébergé sur Kubernetes, fournit une interface unifiée pour tous les consommateurs de données. L'ingestion combine des pipelines par lots pour les extraits des systèmes cœur et des flux CDC en temps réel pour les transactions.

La gouvernance intègre les exigences de la LPRPDE et du Bureau du surintendant des institutions financières (BSIF). Les données sensibles sont classifiées et protégées par des contrôles d'accès granulaires au niveau des colonnes. Le lignage automatisé permet de tracer l'origine de chaque donnée utilisée dans les rapports réglementaires.

Les cas d'usage d'IA incluent la détection de fraude en temps réel (alimentée par les flux CDC), les modèles de risque de crédit (utilisant le time travel pour le backtesting) et les assistants conversationnels pour les conseillers (accédant au lakehouse via RAG).

Bénéfices projetés :

- Réduction de 50 % des coûts de données par rapport aux data warehouses traditionnels
- Délai de mise en marché des nouveaux modèles analytiques réduit de 6 mois à 6 semaines
- Conformité simplifiée grâce au lignage et à l'audit intégrés
- Capacité d'IA différenciante pour l'expérience client

Cas 2 : Détaillant Omnicanal

> **Étude de cas : Entreprise de commerce de détail québécoise (projection 2027)** *Secteur* : Commerce de détail *Contexte* : Détaillant avec présence physique et commerce électronique, données fragmentées entre systèmes de point de vente, plateforme de commerce électronique et CRM *Vision* : Vue unifiée du client et de l'inventaire en quasi-temps réel

Architecture projetée :

Le détaillant implémente un lakehouse Iceberg sur Amazon S3, avec un flux d'intégration Kafka-Iceberg pour les événements de transaction et de navigation. Le streaming lakehouse, tel que décrit dans les chapitres précédents et le Volume III, permet une latence de quelques minutes entre une transaction et sa disponibilité pour l'analytique.

Les domaines de données incluent le client (profil unifié à travers les canaux), l'inventaire (niveaux de stock en quasi-temps réel), les transactions (historique complet avec time travel) et les produits (catalogue et attributs). La fédération avec Trino permet aux analystes d'interroger ces domaines de manière transparente.

L'intégration avec Microsoft Fabric expose les données à Power BI pour les tableaux de bord opérationnels et les analyses ad hoc. Les modèles de prévision de la demande et de personnalisation accèdent aux données via Spark.

Bénéfices projetés :

- Vue client 360° disponible en quelques minutes plutôt que le lendemain
- Optimisation de l'inventaire réduisant les ruptures de stock de 20 %
- Personnalisation en temps réel améliorant le taux de conversion de 15 %
- Flexibilité pour expérimenter de nouveaux cas d'usage analytiques

Cas 3 : Organisme Gouvernemental

> **Étude de cas : Agence fédérale canadienne (projection 2029)** *Secteur* : Gouvernement *Contexte* : Agence avec des obligations de transparence et des contraintes de souveraineté des données *Vision* : Plateforme de données ouverte et sécurisée pour l'analyse de politiques et la publication de données ouvertes

Architecture projetée :

L'agence déploie un lakehouse Iceberg sur une infrastructure infonuagique canadienne certifiée. La séparation stricte entre les zones de données internes et les données destinées à la publication assure la confidentialité. Le catalogue inclut des métadonnées riches pour faciliter la découverte des données tant par les analystes internes que par le public.

Les processus de publication automatisés transforment les données internes en jeux de données ouverts, appliquant l'anonymisation et l'agrégation requises. Le time travel permet de reproduire les analyses historiques pour la vérification et l'audit.

L'intégration avec les systèmes transactionnels gouvernementaux utilise des connecteurs CDC sécurisés. Les données sensibles ne quittent jamais le périmètre contrôlé, mais les analyses agrégées peuvent être partagées.

Bénéfices projetés :

- Conformité avec les directives gouvernementales sur les données ouvertes
- Analyse de politiques accélérée grâce à l'accès unifié aux données
- Traçabilité complète pour l'audit et la reddition de comptes
- Coûts réduits par rapport aux solutions commerciales propriétaires

---

Vision pour l'Avenir du Lakehouse d'Entreprise

Le Lakehouse Autonome

D'ici 2030, le lakehouse évoluera vers un système largement autonome. L'optimisation automatique, la maintenance prédictive et l'auto-réparation réduiront considérablement la charge opérationnelle. Les équipes de données se concentreront sur la modélisation sémantique, la gouvernance et les cas d'usage à valeur ajoutée plutôt que sur les tâches d'infrastructure.

Cette autonomie sera rendue possible par l'apprentissage machine appliqué aux opérations du lakehouse lui-même. Les systèmes apprendront des patrons d'utilisation pour anticiper les besoins, détecter les anomalies et prendre des actions correctives avant que les problèmes n'affectent les utilisateurs.

Le Lakehouse Conversationnel

L'interaction avec le lakehouse deviendra principalement conversationnelle. Les utilisateurs formuleront leurs besoins en langage naturel, et le système traduira ces demandes en requêtes, transformations ou configurations appropriées. Cette démocratisation de l'accès aux données réduira la dépendance envers les compétences techniques spécialisées.

Les assistants IA intégrés au lakehouse fourniront des recommandations contextuelles, expliqueront les résultats des analyses et suggéreront des explorations supplémentaires. Le lakehouse deviendra un partenaire analytique plutôt qu'un simple outil de stockage et de requête.

Le Lakehouse Fédéré

La fédération de lakehouses à travers les organisations deviendra courante. Les entreprises partageront des vues sécurisées de leurs données avec des partenaires, des fournisseurs et des clients sans transfert physique de données. Cette fédération sera facilitée par des standards d'interopérabilité et des protocoles de gouvernance partagés.

Dans le contexte canadien, cette fédération pourrait permettre des initiatives de données sectorielles (par exemple, données de santé anonymisées pour la recherche, données de transport pour la planification urbaine) tout en respectant les exigences de souveraineté et de confidentialité.

Le Lakehouse Agentique

L'intégration étroite entre le lakehouse et les architectures agentiques représente peut-être l'évolution la plus significative. Les agents autonomes, capables d'exécuter des tâches complexes, utiliseront le lakehouse comme mémoire à long terme, source de contexte et destination pour les résultats de leurs actions.

Cette symbiose lakehouse-agents créera de nouvelles possibilités : des agents de support client qui accèdent à l'historique complet des interactions, des agents de gestion de la chaîne d'approvisionnement qui optimisent en continu en fonction des données en temps réel, des agents de conformité qui surveillent automatiquement les violations potentielles.

Le Volume II de cette monographie a établi les fondations de l'infrastructure agentique, et le Volume III a décrit le backbone événementiel Kafka. Le lakehouse Iceberg complète cette architecture en fournissant la couche de persistance nécessaire pour les agents qui nécessitent une mémoire durable et un accès à l'historique.

Résumé

Ce chapitre final a synthétisé les apprentissages du Volume IV et projeté les évolutions du paysage lakehouse pour les cinq prochaines années.

Fondamentaux consolidés : Les quatre piliers d'Iceberg (séparation stockage-calcul, métadonnées sophistiquées, évolution de schéma, partitionnement masqué) restent les fondations de toute architecture lakehouse robuste. Les leçons opérationnelles (maintenance proactive, gouvernance intégrée, observabilité complète) déterminent le succès en production.

État de l'écosystème 2025-2026 : L'adoption par les grands fournisseurs valide Iceberg comme standard de l'industrie. La spécialisation des acteurs et l'interopérabilité comme norme caractérisent le marché actuel. Des défis persistent (petits fichiers, complexité des métadonnées, expertise requise) mais des solutions matures existent.

Tendances 2026-2030 : L'intégration avec l'IA (générative et apprentissage machine) transformera les patrons d'utilisation et d'optimisation du lakehouse. Le format Iceberg évoluera pour supporter de nouveaux types de données et améliorer les performances. La standardisation du catalogue et l'interopérabilité entre formats progresseront. L'intégration avec les architectures agentiques créera de nouvelles possibilités.

Dynamiques de marché : La consolidation se poursuivra, avec une différenciation par la valeur ajoutée. Le contexte canadien (souveraineté, réglementation, écosystème) influence les décisions d'adoption. L'équilibre entre gestion interne et services gérés évoluera vers des modèles hybrides.

Recommandations stratégiques : Les organisations en évaluation devraient commencer par un cas d'usage concret, investir dans les compétences et planifier la gouvernance dès le début. Les organisations en expansion devraient établir un centre d'excellence, automatiser les opérations et préparer l'intégration avec l'IA. Les organisations matures devraient évaluer les capacités avancées, contribuer à l'écosystème et planifier les évolutions architecturales.

Feuille de route : Les quatre phases (fondations, expansion, optimisation, innovation) fournissent un cadre pour structurer le parcours d'adoption sur plusieurs années.

Vision : Le lakehouse évoluera vers un système autonome, conversationnel, fédéré et agentique. Ces évolutions réduiront la charge opérationnelle, démocratiseront l'accès aux données, permettront le partage sécurisé et créeront de nouvelles possibilités d'automatisation intelligente.

Mot de la Fin

Au terme de ce volume, nous espérons avoir fourni aux architectes données, aux ingénieurs de données et aux leaders techniques les connaissances et les outils nécessaires pour concevoir, déployer et opérer un lakehouse Iceberg performant.

Apache Iceberg représente plus qu'un simple format de table : c'est une fondation pour repenser l'architecture des données d'entreprise. En adoptant les principes d'ouverture, de séparation des préoccupations et de gouvernance intégrée, les organisations peuvent construire des plateformes de données agiles, économiques et pérennes.

Les cinq prochaines années apporteront des transformations significatives dans l'écosystème des données. L'intelligence artificielle, les architectures agentiques et les nouvelles formes de collaboration entre organisations redéfiniront les attentes et les possibilités. Le lakehouse Iceberg, par sa conception ouverte et évolutive, est bien positionné pour accompagner ces transformations.

Nous encourageons les lecteurs à commencer leur parcours lakehouse, qu'ils en soient à l'évaluation initiale ou à l'optimisation d'un déploiement existant. Les ressources de la communauté Iceberg, les documentations des fournisseurs et les communautés de pratique offrent un soutien précieux pour naviguer les défis techniques et organisationnels.

Le Volume V de cette monographie, « Le Développeur Renaissance », explorera les compétences et les pratiques nécessaires pour exceller dans ce nouvel environnement technologique. Les ingénieurs de données qui maîtrisent le lakehouse Iceberg, intégré avec le streaming Kafka et les architectures agentiques, seront des contributeurs clés à la transformation numérique de leurs organisations.

Bonne construction de votre lakehouse !

Fin du Chapitre IV.16 - Conclusion Finale et Perspectives 2026-2030

Annexe A - La Spécification Apache Iceberg

Introduction

Cette annexe constitue un guide de référence technique approfondi sur la spécification Apache Iceberg. Elle s'adresse aux architectes données, ingénieurs plateforme et développeurs qui souhaitent comprendre les fondements techniques du format de table Iceberg pour concevoir, opérer et optimiser leurs environnements Lakehouse.

Apache Iceberg est un format de table ouvert conçu pour gérer de vastes collections de fichiers de données dans des systèmes de fichiers distribués ou des magasins objets. La spécification définit précisément comment les métadonnées sont structurées, comment les opérations transactionnelles sont garanties et comment les différents moteurs de requête peuvent interopérer de manière cohérente sur les mêmes tables.

La compréhension de cette spécification est essentielle pour quiconque souhaite tirer pleinement parti des capacités d'Iceberg, que ce soit pour l'évolution de schéma, le partitionnement masqué, le voyage dans le temps ou l'optimisation des performances de requête.

A.1 Comprendre la spécification Iceberg

A.1.1 Philosophie et principes fondamentaux

Apache Iceberg a émergé en 2017 chez Netflix pour résoudre un problème fondamental : les lacs de données construits sur des formats de fichiers comme Apache Parquet et ORC manquaient de garanties transactionnelles, d'évolution de schéma fiable et de visibilité sur l'état des tables. Le format Hive, largement utilisé à l'époque, montrait ses limites, particulièrement à grande échelle et lors de la migration vers des magasins objets infonuagiques comme Amazon S3.

La spécification Iceberg repose sur plusieurs principes directeurs :

Séparation logique et physique. La définition logique d'une table est découplée de sa disposition physique en stockage. Cette séparation permet à plusieurs moteurs de calcul d'opérer sur le même jeu de données avec des garanties transactionnelles complètes, sans dépendre de conventions de répertoires ou de systèmes de fichiers spécifiques.

Immutabilité des fichiers. Les fichiers de données ne sont jamais modifiés après leur écriture. Les mises à jour et suppressions créent de nouveaux fichiers ou des fichiers de suppression, tandis que les snapshots maintiennent l'historique des états de la table. Cette immutabilité simplifie considérablement la gestion de la concurrence et permet des fonctionnalités comme le voyage dans le temps.

Métadonnées riches. Iceberg maintient des statistiques détaillées au niveau des colonnes, des partitions et des fichiers. Ces métadonnées permettent aux moteurs de requête d'élaguer efficacement les fichiers et partitions non pertinents, réduisant drastiquement les données analysées.

Transactions ACID. Chaque modification de table est atomique. Les lecteurs voient toujours un état cohérent de la table, même pendant les écritures concurrentes. Les conflits sont détectés et résolus au moment du commit grâce au contrôle optimiste de la concurrence.

A.1.2 Architecture en couches

La spécification définit une architecture en trois couches distinctes :

Couche catalogue. Le catalogue maintient une référence vers le fichier de métadonnées courant pour chaque table. Cette référence est le point d'entrée unique pour accéder à une table Iceberg. Le catalogue peut être implémenté de diverses manières : Hive Metastore, catalogue REST, JDBC, AWS Glue, ou fichiers dans un système de stockage.

Couche métadonnées. Cette couche comprend le fichier de métadonnées JSON (metadata.json), les listes de manifestes (manifest lists) et les fichiers de manifestes (manifest files). Le fichier de métadonnées contient le schéma de la table, les spécifications de partitionnement, les propriétés et la liste des snapshots. Chaque snapshot référence une liste de manifestes qui elle-même pointe vers les fichiers de manifestes décrivant les fichiers de données.

Couche données. Les fichiers de données contiennent les enregistrements réels dans des formats ouverts comme Parquet, ORC ou Avro. Les fichiers de suppression (delete files) et vecteurs de suppression (deletion vectors) marquent les lignes supprimées sans réécrire les fichiers de données originaux.

A.1.3 Structure des métadonnées

Le fichier de métadonnées JSON constitue le cœur de la définition d'une table Iceberg. Il contient :

Informations de version. Le numéro de version du format (1, 2 ou 3), l'UUID unique de la table, et l'emplacement du fichier de métadonnées lui-même.

Schéma. La définition complète des colonnes avec leurs types, identifiants uniques et informations de documentation. Les identifiants de champs sont stables à travers les évolutions de schéma, permettant la compatibilité ascendante et descendante.

Spécifications de partition. Les règles définissant comment les données sont partitionnées, incluant les transformations appliquées aux colonnes sources (identity, bucket, truncate, year, month, day, hour).

Spécifications de tri. L'ordre dans lequel les données sont triées au sein des fichiers, optimisant certains types de requêtes.

Snapshots. La liste des états historiques de la table, chaque snapshot représentant un état complet et cohérent. Le snapshot courant définit l'état visible par défaut.

Historique des snapshots. Le journal des changements de snapshot courant, permettant le voyage dans le temps vers n'importe quel état précédent.

Les fichiers de manifestes sont des fichiers Avro contenant des métadonnées détaillées sur les fichiers de données : chemins, tailles, statistiques de colonnes (min, max, compte de null, compte de valeurs distinctes), et informations de partition.

A.1.4 Garanties transactionnelles

Iceberg implémente l'isolation des snapshots (snapshot isolation) pour garantir la cohérence des lectures et écritures concurrentes :

Lectures cohérentes. Une lecture voit toujours un état complet de la table correspondant à un snapshot spécifique. Les modifications en cours n'affectent pas les lectures existantes.

Écritures atomiques. Une écriture crée un nouveau snapshot contenant tous les changements. L'opération réussit entièrement ou échoue entièrement, sans état intermédiaire visible.

Contrôle optimiste de concurrence. Les écritures concurrentes sont détectées au moment du commit. En cas de conflit, l'écriture peut être réessayée avec une base plus récente. Les conflits sont résolus au niveau du catalogue, qui garantit qu'un seul commit réussit pour un état donné.

Sérialisation des modifications. Les modifications sont sérialisées par le catalogue via des mécanismes de verrouillage ou de comparaison-et-échange (compare-and-swap), assurant la linéarisabilité des opérations.

A.2 Versions du format de table Iceberg

A.2.1 Version 1 : Les fondations

La version 1, première version stable de la spécification, établit les fondations de la gestion de tables analytiques à grande échelle. Elle définit :

Gestion des fichiers immuables. Support des formats Parquet, Avro et ORC pour le stockage des données. Les fichiers sont écrits une fois et jamais modifiés, garantissant l'intégrité des données et simplifiant la réplication.

Métadonnées structurées. Architecture en couches avec fichiers de métadonnées JSON, listes de manifestes et manifestes Avro. Cette structure permet une navigation efficace de la table sans lister récursivement les répertoires.

Évolution de schéma. Support pour l'ajout, la suppression et le renommage de colonnes sans réécriture des données existantes. Les identifiants de champs stables assurent la compatibilité.

Évolution de partition. Possibilité de modifier le schéma de partitionnement sans réécrire les données. Les nouvelles données utilisent le nouveau schéma tandis que les anciennes conservent leur partitionnement original.

Partitionnement masqué. Les colonnes de partition ne sont pas exposées dans le schéma de la table. Les transformations de partition sont appliquées automatiquement lors des écritures et des requêtes.

La version 1 est conçue pour des charges de travail principalement en ajout (append-only), typiques des entrepôts de données traditionnels et des lacs de données analytiques.

A.2.2 Version 2 : Suppressions au niveau des lignes

La version 2, introduite pour supporter des charges de travail plus dynamiques, ajoute des capacités essentielles pour les pipelines de données modernes :

Fichiers de suppression par position. Les fichiers de suppression par position identifient les lignes supprimées par leur chemin de fichier de données et leur position de ligne dans ce fichier. Cette approche évite la réécriture des fichiers de données lors de suppressions ponctuelles.

Fichiers de suppression par égalité. Les fichiers de suppression par égalité identifient les lignes supprimées par les valeurs d'une ou plusieurs colonnes. Cette approche est particulièrement efficace pour les suppressions basées sur des clés métier.

Numéros de séquence. Chaque opération reçoit un numéro de séquence croissant, permettant d'ordonner les opérations et de déterminer quels fichiers de suppression s'appliquent à quels fichiers de données. Un fichier de suppression ne s'applique qu'aux fichiers de données avec un numéro de séquence inférieur ou égal.

Modes Copy-on-Write et Merge-on-Read. Les utilisateurs peuvent choisir entre réécrire les fichiers de données lors des modifications (Copy-on-Write, optimisé pour la lecture) ou créer des fichiers de suppression réconciliés à la lecture (Merge-on-Read, optimisé pour l'écriture).

La version 2 rend Iceberg adapté aux charges de travail dynamiques avec mises à jour fréquentes, aux pipelines de capture de données modifiées (CDC), et aux exigences de conformité RGPD nécessitant des suppressions ciblées.

A.2.3 Version 3 : Performance et expressivité

La version 3, ratifiée par la communauté Apache Iceberg en 2025, représente un saut significatif en termes de performance et de flexibilité. Les premières fonctionnalités V3 sont apparues dans Iceberg 1.8.0 (février 2025), avec des ajouts dans les versions 1.9.0 et 1.10.0.

Vecteurs de suppression (Deletion Vectors). Les vecteurs de suppression remplacent les fichiers de suppression par position avec un format binaire compact stocké dans des fichiers Puffin. Utilisant des bitmaps Roaring, ils offrent une représentation plus efficace des suppressions et éliminent la traduction entre fichiers Parquet et représentations mémoire. Les vecteurs de suppression sont liés à un seul fichier de données, avec au plus un vecteur par fichier de données dans un snapshot.

Lignage des lignes (Row Lineage). La version 3 introduit des métadonnées au niveau des lignes pour simplifier le traitement incrémental. Chaque ligne reçoit un identifiant unique (`_row_id`) et un numéro de séquence de dernière mise à jour (`_last_updated_sequence_number`). Ces champs permettent aux moteurs de détecter les changements au niveau des lignes entre commits, optimisant les vues matérialisées et les pipelines CDC.

Valeurs par défaut des colonnes. Les colonnes peuvent désormais avoir des valeurs par défaut définies dans les métadonnées. L'ajout d'une colonne avec valeur par défaut est instantané (modification des métadonnées uniquement) sans réécriture des fichiers de données existants. Les moteurs de requête appliquent la valeur par défaut à la volée pour les anciens fichiers.

Types de données avancés. La version 3 ajoute plusieurs nouveaux types : - **Variant** : Type flexible pour données semi-structurées (similaire à JSON) avec support pour date, timestamp, binary et decimal - **Geometry et Geography** : Types géospatiaux pour analyses de localisation - **Timestamps nanosecondes** : Précision nanoseconde avec ou sans fuseau horaire

Transformations multi-arguments. Support pour des stratégies de partitionnement et de tri plus avancées utilisant plusieurs colonnes ou fonctions composites.

A.2.4 Compatibilité entre versions

La spécification Iceberg est conçue pour la compatibilité ascendante et descendante :

Compatibilité ascendante. Les lecteurs d'une version ultérieure peuvent lire les tables de versions antérieures. Les fichiers de métadonnées V1 sont valides dans les tables V2 et V3, permettant les mises à niveau sans réécriture de l'arbre de métadonnées.

Mise à niveau atomique. Une table peut être mise à niveau vers une version ultérieure de manière atomique. Par exemple, la mise à niveau de V2 vers V3 crée un nouveau snapshot de métadonnées, réutilise les fichiers de données Parquet existants, et ajoute les champs de lignage des lignes aux métadonnées de table.

Mise à niveau irréversible. Les mises à niveau de version sont unidirectionnelles. Une table mise à niveau de V2 vers V3 ne peut pas être rétrogradée vers V2 via des opérations standard.

Tolérance des lecteurs. Les lecteurs doivent être tolérants envers les champs inconnus et les transformations inconnues pour assurer la compatibilité future. Les champs de partition avec transformations inconnues sont ignorés lors de la planification des scans.

A.3 Gestion des snapshots et métadonnées

A.3.1 Anatomie d'un snapshot

Un snapshot représente un état complet et cohérent d'une table Iceberg à un instant donné. Chaque snapshot contient :

Identifiant unique. Un entier long identifiant de manière unique le snapshot au sein de la table.

ID de snapshot parent. Référence au snapshot précédent, formant une chaîne de lignage permettant le voyage dans le temps.

Numéro de séquence. Numéro croissant assigné lors du commit, utilisé pour ordonner les opérations et déterminer l'applicabilité des fichiers de suppression.

Horodatage. Moment de création du snapshot en millisecondes depuis l'époque Unix.

Emplacement de la liste de manifestes. Chemin vers le fichier de liste de manifestes décrivant les fichiers de données et de suppression du snapshot.

Résumé. Métadonnées résumant les changements : opération effectuée (append, replace, overwrite, delete), statistiques de modifications (fichiers ajoutés/supprimés, enregistrements ajoutés/supprimés).

A.3.2 Structure des manifestes

La liste de manifestes (manifest list) est un fichier Avro contenant des métadonnées sur les fichiers de manifestes du snapshot :

- Chemin du fichier de manifeste
- Longueur du fichier
- ID de la spécification de partition utilisée
- Plages de valeurs de partition couvertes
- Statistiques agrégées : nombre de fichiers, nombre d'enregistrements, comptes de suppressions

Les fichiers de manifestes (manifest files) sont également des fichiers Avro contenant des métadonnées détaillées sur les fichiers de données :

Informations de fichier. Chemin, format (Parquet/ORC/Avro), taille en octets, nombre d'enregistrements.

Statistiques de colonnes. Pour chaque colonne : valeurs minimale et maximale, compte de valeurs null, compte de valeurs NaN (pour les flottants).

Informations de partition. Valeurs des colonnes de partition pour le fichier.

Métadonnées de tri. Ordre de tri des données dans le fichier, permettant une fusion efficace lors des lectures.

A.3.3 Opérations sur les snapshots

Les opérations de base créant de nouveaux snapshots incluent :

Append. Ajout de nouveaux fichiers de données sans modifier les fichiers existants. L'opération la plus courante pour les charges de travail d'ingestion.

Overwrite. Remplacement de fichiers de données correspondant à un prédicat de filtre. Utilisé pour les réécritures partielles de table.

Replace. Remplacement complet des fichiers de la table. Utilisé pour les reconstructions complètes ou les opérations de compaction.

Delete. Ajout de fichiers de suppression marquant des lignes comme supprimées. Utilisé en mode Merge-on-Read.

Row Delta. Combinaison d'ajouts de fichiers de données et de fichiers de suppression dans une même opération atomique.

A.3.4 Voyage dans le temps

Le voyage dans le temps (Time Travel) permet d'interroger la table à un état historique spécifique :

Par identifiant de snapshot. Interrogation d'un snapshot spécifique par son ID unique.

Par horodatage. Interrogation de l'état de la table à un moment donné. Iceberg sélectionne le snapshot le plus récent dont l'horodatage est antérieur ou égal à la valeur spécifiée.

Syntaxe SQL. La plupart des moteurs supportent la syntaxe `AS OF` pour le voyage dans le temps :

```
SELECT * FROM table AS OF TIMESTAMP '2025-01-01 00:00:00';
SELECT * FROM table AS OF VERSION 12345678901234;
```

A.3.5 Expiration des snapshots

L'expiration des snapshots libère l'espace de stockage en supprimant les métadonnées et fichiers de données obsolètes :

Politique de rétention. Les snapshots plus anciens qu'un seuil configurable sont candidats à l'expiration. La politique par défaut conserve généralement quelques jours ou semaines d'historique.

Fichiers orphelins. L'expiration supprime les fichiers de données qui ne sont plus référencés par aucun snapshot retenu.

Opération de maintenance. L'expiration doit être exécutée régulièrement pour éviter l'accumulation de fichiers obsolètes et la croissance incontrôlée du stockage.

A.3.6 Rollback

Le rollback permet de revenir à un état antérieur de la table :

Rollback à un snapshot. Définit un snapshot historique comme le snapshot courant de la table.

Préservation de l'historique. Le rollback ne supprime pas les snapshots intermédiaires. Les nouvelles écritures après rollback créent des snapshots basés sur l'état restauré.

Branches et tags. Iceberg supporte des branches et tags nommés pour référencer des snapshots spécifiques, facilitant les workflows de gestion de versions des données.

A.4 La spécification REST Catalog

A.4.1 Motivation et objectifs

La spécification REST Catalog a été développée pour résoudre les problèmes de compatibilité et d'interopérabilité entre catalogues. Avant son introduction, chaque implémentation de catalogue nécessitait des clients spécifiques dans chaque langage supporté par Iceberg (Java, Python, Rust, Go), créant des incohérences et des barrières à l'adoption.

Les objectifs principaux de la spécification REST Catalog sont :

Compatibilité universelle. Une seule implémentation client peut interagir avec n'importe quel catalogue conforme à la spécification, indépendamment de l'implémentation serveur.

Flexibilité d'implémentation. Les fournisseurs de catalogues peuvent implémenter le serveur dans n'importe quel langage et utiliser n'importe quelle technologie de stockage, tant que l'API REST est respectée.

Centralisation des opérations. Les opérations sont gérées côté serveur plutôt que côté client, permettant un contrôle centralisé et des mises à jour simplifiées.

Sécurité intégrée. Support natif pour OAuth 2.0 et les mécanismes d'authentification standard.

A.4.2 Structure de l'API

La spécification REST Catalog définit une API OpenAPI couvrant les opérations suivantes :

Gestion des espaces de noms. Création, listage, mise à jour et suppression d'espaces de noms (équivalents aux bases de données ou schémas).

```
GET    /v1/{prefix}/namespaces
POST   /v1/{prefix}/namespaces
GET    /v1/{prefix}/namespaces/{namespace}
DELETE /v1/{prefix}/namespaces/{namespace}
POST   /v1/{prefix}/namespaces/{namespace}/properties
```

Gestion des tables. Création, listage, chargement, mise à jour et suppression de tables.

```
GET    /v1/{prefix}/namespaces/{namespace}/tables
POST   /v1/{prefix}/namespaces/{namespace}/tables
GET    /v1/{prefix}/namespaces/{namespace}/tables/{table}
POST   /v1/{prefix}/namespaces/{namespace}/tables/{table}
DELETE /v1/{prefix}/namespaces/{namespace}/tables/{table}
HEAD   /v1/{prefix}/namespaces/{namespace}/tables/{table}
POST   /v1/{prefix}/namespaces/{namespace}/tables/{table}/metrics
```

Gestion des vues. Support des vues Iceberg avec opérations similaires aux tables.

Commit des transactions. Endpoint pour soumettre les changements de métadonnées avec résolution des conflits côté serveur.

A.4.3 Authentification et sécurité

La spécification REST Catalog intègre OAuth 2.0 pour l'authentification :

Flux client credentials. Échange de credentials (client ID et secret) contre un jeton d'accès.

Échange de jetons. Échange d'un jeton utilisateur (jeton sujet) contre un jeton d'accès plus spécifique, utilisant le jeton d'accès du catalogue comme jeton acteur.

Jetons bearer. Les requêtes aux tables et vues peuvent inclure des jetons d'autorisation bearer si la sécurité OAuth2 est activée.

Signature S3 distante. Pour le stockage S3, la spécification permet la signature distante des requêtes, centralisant les credentials côté serveur.

A.4.4 Configuration et credentials de stockage

Le REST Catalog gère la configuration d'accès au stockage :

Storage credentials. Le serveur retourne des credentials temporaires pour accéder aux fichiers de données. Les clients vérifient d'abord le champ `storage-credentials` avant de consulter le champ `config`.

Configuration FileIO. Le serveur peut spécifier une implémentation FileIO spécifique pour la table selon son stockage sous-jacent.

Préfixes de credentials. Support pour plusieurs préfixes de credentials de stockage, permettant l'accès à des fichiers dans différentes régions ou comptes.

A.4.5 Implémentations notables

Plusieurs implémentations de la spécification REST Catalog sont disponibles :

Apache Polaris (Incubating). Catalogue open source développé initialement par Snowflake, offrant un serveur REST Catalog complet avec contrôle d'accès granulaire.

Project Nessie. Catalogue Git-like avec versioning des métadonnées, branches et merges, exposant une API REST Catalog compatible.

AWS Glue Data Catalog. Service AWS géré offrant une API REST Catalog native pour l'intégration avec les services AWS et Iceberg.

Google BigLake REST Catalog. Implémentation Google Cloud entièrement gérée, intégrée avec BigQuery pour la gouvernance et la sécurité.

Dremio Arctic / Hybrid Catalog. Catalogue géré Dremio intégrant maintenance automatique et fédération multi-emplacements.

A.4.6 Évolutions futures

La spécification REST Catalog continue d'évoluer pour supporter de nouveaux cas d'usage :

Endpoint de planification de scan. Un endpoint permettant de déléguer la planification des scans au serveur est en discussion. Cela permettrait aux moteurs de requête de bénéficier d'optimisations centralisées et de compatibilité multi-formats.

Service de maintenance. Un système de notification pour les requêtes de maintenance permettrait aux outils souscrits d'optimiser automatiquement les tables selon des conditions configurables.

Fédération de catalogues. Support pour l'interrogation de tables à travers plusieurs catalogues, simplifiant la gestion d'écosystèmes de données distribués.

A.5 Spécification du format de fichier Puffin

A.5.1 Objectif et cas d'usage

Puffin est un format de fichier conteneur conçu pour stocker des informations auxiliaires sur les données gérées par Iceberg qui ne peuvent pas être stockées directement dans les manifestes. Le format a été adopté par la communauté Apache Iceberg en 2022 et est devenu central avec l'introduction des vecteurs de suppression en V3.

Les cas d'usage principaux de Puffin incluent :

Statistiques de table. Stockage d'esquisses (sketches) pour estimer le nombre de valeurs distinctes (NDV) d'une colonne. Ces statistiques sont essentielles pour l'optimisation des plans de requête, notamment le réordonnancement des jointures.

Vecteurs de suppression. En V3, les vecteurs de suppression sont stockés dans des fichiers Puffin comme bitmaps binaires compacts, remplaçant les fichiers de suppression par position de V2.

Index secondaires. Le format est conçu pour accueillir des index comme les filtres Bloom ou autres structures d'accès accéléré.

A.5.2 Structure du fichier

Un fichier Puffin suit une structure simple et efficace :

```
Magic (4 octets) | Blob₁ | Blob₂ | ... | Blobₙ | Footer
```

Magic. Quatre octets identifiant le format : `0x50, 0x46, 0x41, 0x31` (P, F, A, 1 en ASCII).

Blobs. Séquence de blobs binaires contenant les données réelles (statistiques, vecteurs de suppression, index).

Footer. Métadonnées JSON décrivant les blobs contenus dans le fichier.

A.5.3 Métadonnées de fichier

Le footer contient un objet JSON `FileMetadata` avec :

Blobs. Liste des métadonnées de chaque blob : - `type` : Type du blob (ex. `apache-datasketches-theta-v1`, `deletion-vector-v1`) - `fields` : Liste des IDs de champs concernés - `snapshot-id` : ID du snapshot source - `sequence-number` : Numéro de séquence du snapshot - `offset` : Position du blob dans le fichier - `length` : Taille du blob en octets - `compression-codec` : Codec de compression utilisé (optionnel)

Properties. Propriétés générales du fichier : - `created-by` : Identification de l'application ayant créé le fichier et sa version

A.5.4 Types de blobs définis

La spécification définit plusieurs types de blobs :

apache-datasketches-theta-v1. Esquisse Theta sérialisée produite par la bibliothèque Apache DataSketches. L'esquisse est construite en utilisant la famille Alpha avec la graine par défaut, alimentée avec les valeurs distinctes converties en octets selon la sérialisation single-value d'Iceberg.

Les métadonnées du blob peuvent inclure : - `ndv` : Estimation du nombre de valeurs distinctes dérivée de l'esquisse

deletion-vector-v1. Vecteur de suppression sérialisé représentant les positions des lignes supprimées dans un fichier de données. Un bit à 1 à la position P indique que la ligne à la position P est supprimée.

Le format utilise des bitmaps Roaring pour une représentation compacte. Les métadonnées requises incluent : - `referenced-data-file` : Emplacement du fichier de données référencé - `cardinality` : Nombre de lignes supprimées (bits à 1) dans le vecteur

Les vecteurs de suppression ne sont pas compressés (le champ `compression-codec` est omis).

A.5.5 Compression

Les blobs peuvent être compressés avec les codecs suivants pour une interopérabilité maximale :

Codec	Description
`lz4`	Frame de compression LZ4 unique avec taille de contenu présente
`zstd`	Frame de compression Zstandard unique avec taille de contenu présente

Le footer JSON lui-même peut être compressé en LZ4. Les données peuvent également être non compressées si le codec n'est pas spécifié.

A.5.6 Intégration avec Iceberg

Les fichiers Puffin sont référencés dans les métadonnées de table Iceberg :

Fichiers de statistiques. Les fichiers Puffin contenant des statistiques sont enregistrés dans le champ `statistics` des métadonnées de table. Chaque entrée contient : - `snapshot-id` : ID du snapshot associé - `statistics-path` : Chemin du fichier Puffin - `file-size-in-bytes` : Taille du fichier - `file-footer-size-in-bytes` : Taille du footer - `blob-metadata` : Liste des métadonnées de blobs

Vecteurs de suppression. Les vecteurs de suppression sont suivis individuellement dans les manifestes de suppression par emplacement de fichier, offset et longueur dans le fichier Puffin conteneur.

A.5.7 Considérations de performance

L'utilisation de fichiers Puffin offre plusieurs avantages de performance :

Planification de requêtes optimisée. Les statistiques NDV permettent aux optimiseurs de choisir les meilleures stratégies de jointure et d'ordonnancement des opérations.

Prédiction d'élagage améliorée. Les statistiques détaillées permettent un pushdown de prédicats plus efficace, réduisant les données scannées.

Suppressions efficaces. Les vecteurs de suppression évitent l'amplification d'écriture des mises à jour par lots et des suppressions de conformité, réduisant significativement le surcoût de maintenance des données fraîches.

Traitement incrémental. Les esquisses peuvent être mises à jour de manière incrémentale sans relire les données complètes de la table.

A.6 Compatibilité et migration

A.6.1 Stratégies de migration vers Iceberg

La migration vers Apache Iceberg peut suivre plusieurs approches selon le contexte :

Migration in-place. Conversion d'une table existante (Hive, par exemple) vers Iceberg en place. Cette approche modifie les métadonnées de la table originale pour la convertir en table Iceberg, préservant l'emplacement des fichiers de données.

```
CALL spark_catalog.system.migrate('hive_db.hive_table')
```

Migration par snapshot. Création d'une copie Iceberg d'une table existante tout en préservant la table source. Cette approche permet de valider la migration avant de basculer les applications.

```
CALL spark_catalog.system.snapshot(
  source_table => 'hive_db.hive_table',
  table => 'iceberg_db.iceberg_table'
)
```

Migration avec reconfiguration. Migration vers Iceberg avec modification simultanée des propriétés de table (version de format, modes de suppression, etc.).

```
CALL spark_catalog.system.snapshot(
  source_table => 'hive_db.hive_table',
  table => 'iceberg_db.iceberg_table',
  properties => map(
    'format-version', '3',
    'write.delete.mode', 'merge-on-read'
```

```
  )
)
```

A.6.2 Mise à niveau des versions de format

La mise à niveau entre versions de format Iceberg suit des règles spécifiques :

De V1 vers V2. La mise à niveau active le support des suppressions au niveau des lignes. Elle est effectuée en modifiant la propriété de table :

```
ALTER TABLE my_table SET TBLPROPERTIES('format-version' = '2')
```

De V2 vers V3. La mise à niveau active les vecteurs de suppression et le lignage des lignes. Elle est atomique et ne nécessite pas de réécriture des données :

```
ALTER TABLE my_table SET TBLPROPERTIES('format-version' = '3')
```

Après mise à niveau vers V3 : - Les champs de lignage des lignes sont ajoutés aux métadonnées de table - La prochaine compaction supprimera les anciens fichiers de suppression V2 - Les nouvelles modifications utiliseront les fichiers de vecteurs de suppression V3 - La mise à niveau ne remplit pas rétroactivement les enregistrements de suivi de lignage

A.6.3 Compatibilité des moteurs

Avant de migrer ou mettre à niveau, il est essentiel de vérifier la compatibilité des moteurs :

Moteur	V1	V2	V3
Apache Spark 3.5+	✓	✓	✓ (avec Iceberg 1.8+)
Apache Flink 2.0+	✓	✓	✓ (avec Iceberg 1.10+)
Trino	✓	✓	Partiel
Dremio	✓	✓	✓
AWS Athena	✓	✓	En cours
Snowflake	✓	✓	Annoncé

La vérification de compatibilité doit inclure : - Support des types de données utilisés - Support des modes de suppression requis - Performance acceptable pour les cas d'usage cibles

A.6.4 Considérations de rétrocompatibilité

Lors de la planification d'une migration ou mise à niveau :

Tables multi-moteurs. Si plusieurs moteurs accèdent à la même table, tous doivent supporter la version cible avant la mise à niveau.

Applications héritées. Les applications non compatibles avec les nouvelles versions nécessitent soit une mise à jour, soit le maintien de tables dans l'ancienne version.

Période de transition. Prévoir une période où les deux versions coexistent pour valider le comportement avant de compléter la migration.

A.6.5 Rollback de migration

En cas de problème après migration :

Rollback de données. Utiliser le voyage dans le temps Iceberg pour revenir à un état antérieur si les données sont corrompues.

Rollback de version. La rétrogradation de version (V3 vers V2) n'est pas supportée nativement. La solution consiste à recréer la table en V2 à partir d'un export ou d'un snapshot antérieur.

Tables de sauvegarde. Conserver les tables sources pendant la période de validation pour permettre un retour arrière complet si nécessaire.

A.6.6 Meilleures pratiques de migration

Pour une migration réussie, les pratiques suivantes sont recommandées :

Environnement de test. Valider la migration dans un environnement de développement avant la production.

Migration incrémentale. Commencer par les tables les moins critiques pour acquérir de l'expérience avant de migrer les tables stratégiques.

Monitoring post-migration. Surveiller les performances, la taille des fichiers et les métriques de requête après migration pour détecter les problèmes.

Documentation. Documenter les configurations, les décisions et les procédures de migration pour faciliter les futures évolutions.

Résumé

Cette annexe a présenté les fondements techniques de la spécification Apache Iceberg, couvrant :

Principes fondamentaux. L'architecture en couches d'Iceberg sépare la définition logique des tables de leur stockage physique, permettant l'interopérabilité entre moteurs et les garanties transactionnelles ACID sur des systèmes de fichiers distribués et des magasins objets.

Versions du format. La spécification a évolué à travers trois versions majeures : V1 établissant les fondations pour les charges de travail analytiques, V2 ajoutant les suppressions au niveau des lignes pour les pipelines dynamiques, et V3 introduisant les vecteurs de suppression, le lignage des lignes et de nouveaux types de données pour une performance et une expressivité accrues.

Gestion des snapshots. Le système de snapshots immutables d'Iceberg permet le voyage dans le temps, le rollback et la concurrence optimiste, tout en nécessitant une maintenance régulière via l'expiration des snapshots pour contrôler la croissance du stockage.

REST Catalog. La spécification REST Catalog standardise l'interaction avec les catalogues Iceberg, permettant l'interopérabilité universelle entre clients et serveurs indépendamment de leur implémentation.

Format Puffin. Le format de fichier Puffin stocke les statistiques avancées et les vecteurs de suppression, complétant les métadonnées des manifestes pour des optimisations de requête et des suppressions plus efficaces.

Migration et compatibilité. Les stratégies de migration permettent l'adoption progressive d'Iceberg et la mise à niveau entre versions de format, tout en maintenant la compatibilité avec les écosystèmes existants.

La maîtrise de cette spécification est essentielle pour concevoir des architectures Lakehouse robustes, optimiser les performances des requêtes et opérer efficacement des environnements de données à grande échelle.

Dernière mise à jour : Janvier 2026 Basé sur la spécification Apache Iceberg v3 et les versions 1.8.0 à 1.10.x

Annexe B - Glossaire

Introduction

Ce glossaire rassemble les termes techniques essentiels utilisés dans le Volume IV et plus largement dans l'écosystème Apache Iceberg et Data Lakehouse. Il constitue un outil de référence normalisé pour assurer une compréhension cohérente de la terminologie à travers l'ensemble de la monographie.

Les définitions sont organisées en quatre sections thématiques : terminologie Apache Iceberg, terminologie Lakehouse et Data Engineering, terminologie Streaming et Kafka, et acronymes et abréviations. Pour chaque terme, nous fournissons une définition concise, des exemples d'utilisation lorsque pertinent, et des références croisées vers les termes connexes.

B.1 Terminologie Apache Iceberg

Catalog (Catalogue)

Service ou composant maintenant les références vers les fichiers de métadonnées courants des tables Iceberg. Le catalogue sert de point d'entrée unique pour accéder aux tables et garantit l'atomicité des commits via des mécanismes de verrouillage ou de compare-and-swap.

Exemples : REST Catalog, Hive Metastore Catalog, AWS Glue Catalog, JDBC Catalog, Apache Polaris

Voir aussi : REST Catalog, Metastore

Compaction

Processus de maintenance consolidant plusieurs petits fichiers de données en fichiers plus grands et optimisés. La compaction améliore les performances de lecture en réduisant le nombre de fichiers à scanner et en optimisant la disposition des données selon les patterns d'accès.

Types : - **Bin-packing** : Regroupement de fichiers sans réorganisation du contenu - **Sort compaction** : Réorganisation et tri des données selon les colonnes fréquemment utilisées

Voir aussi : Maintenance, Rewrite Data Files

Copy-on-Write (CoW)

Mode d'écriture où les modifications de données entraînent la réécriture complète des fichiers affectés. Une mise à jour d'une seule ligne nécessite la réécriture du fichier entier contenant cette ligne, créant une nouvelle version avec la modification appliquée.

Avantages : Lecture optimale sans réconciliation *Inconvénients* : Amplification d'écriture significative

Voir aussi : Merge-on-Read, Write Amplification

Data File (Fichier de données)

Fichier contenant les enregistrements réels d'une table Iceberg, stocké dans un format ouvert comme Parquet, ORC ou Avro. Les fichiers de données sont immuables après écriture et référencés par les fichiers de manifestes.

Voir aussi : Manifest File, Delete File

Delete File (Fichier de suppression)

Fichier marquant des lignes comme supprimées sans réécrire les fichiers de données originaux. Utilisé en mode Merge-on-Read pour des suppressions et mises à jour efficaces.

Types : - **Position delete file** : Identifie les lignes par chemin de fichier et position (V2, déprécié en V3) - **Equality delete file** : Identifie les lignes par valeurs de colonnes

Voir aussi : Deletion Vector, Merge-on-Read

Deletion Vector (Vecteur de suppression)

Structure binaire compacte introduite en V3 identifiant les lignes supprimées dans un fichier de données par leur position, stockée sous forme de bitmap Roaring dans un fichier Puffin. Plus efficace que les fichiers de suppression par position de V2.

Caractéristiques : - Un seul vecteur par fichier de données dans un snapshot - Format bitmap pour représentation compacte - Stocké dans des fichiers Puffin

Voir aussi : Delete File, Puffin, Roaring Bitmap

Equality Delete (Suppression par égalité)

Méthode de suppression identifiant les lignes à supprimer par les valeurs d'une ou plusieurs colonnes clés, plutôt que par leur position dans un fichier. Particulièrement utile pour les pipelines CDC où la clé primaire identifie les enregistrements.

Voir aussi : Delete File, Position Delete

Hidden Partitioning (Partitionnement masqué)

Fonctionnalité d'Iceberg où les colonnes de partition ne sont pas exposées dans le schéma de la table. Les transformations de partition sont définies séparément et appliquées automatiquement, évitant les erreurs de requête liées aux colonnes de partition.

Transformations supportées : identity, bucket, truncate, year, month, day, hour

Voir aussi : Partition Evolution, Partition Spec

Manifest File (Fichier de manifeste)

Fichier Avro contenant les métadonnées détaillées des fichiers de données : chemins, tailles, statistiques de colonnes (min, max, nulls, NaN), informations de partition et métadonnées de tri. Les manifestes permettent l'élagage efficace sans lister le stockage.

Voir aussi : Manifest List, Data File

Manifest List (Liste de manifestes)

Fichier Avro listant les fichiers de manifestes d'un snapshot avec des métadonnées résumées : plages de partition, statistiques agrégées et compteurs de fichiers. Permet l'élagage rapide au niveau des manifestes.

Voir aussi : Manifest File, Snapshot

Merge-on-Read (MoR)

Mode d'écriture où les modifications créent des fichiers de suppression réconciliés avec les données originales au moment de la lecture. Les mises à jour et suppressions sont appliquées pendant l'exécution des requêtes.

Avantages : Écriture rapide sans amplification *Inconvénients* : Surcoût de lecture pour la réconciliation

Voir aussi : Copy-on-Write, Delete File

Metadata File (Fichier de métadonnées)

Fichier JSON contenant la définition complète d'une table Iceberg : schéma, spécifications de partition, propriétés, snapshots, historique et statistiques. Le catalogue pointe vers le fichier de métadonnées courant.

Contenu : format-version, table-uuid, location, schemas, partition-specs, sort-orders, current-snapshot-id, snapshots, snapshot-log

Voir aussi : Snapshot, Schema Evolution

Partition Evolution (Évolution de partition)

Capacité de modifier le schéma de partitionnement sans réécrire les données existantes. Les nouvelles données utilisent le nouveau schéma tandis que les anciennes conservent leur partitionnement original, avec réconciliation automatique lors des requêtes.

Voir aussi : Hidden Partitioning, Schema Evolution

Partition Pruning (Élagage de partition)

Optimisation excluant les partitions non pertinentes de la planification de scan basée sur les prédicats de requête. L'élagage utilise les informations de partition stockées dans les manifestes sans accéder aux fichiers de données.

Voir aussi : File Pruning, Statistics

Position Delete (Suppression par position)

Méthode de suppression identifiant les lignes par leur chemin de fichier de données et leur position de ligne (offset) dans ce fichier. Dépréciée en V3 au profit des vecteurs de suppression.

Voir aussi : Delete File, Deletion Vector

Puffin

Format de fichier conteneur pour stocker des informations auxiliaires sur les données Iceberg : statistiques, index et vecteurs de suppression. Structure simple avec magic bytes, blobs binaires et footer JSON.

Cas d'usage : - Esquisses Theta pour statistiques NDV - Vecteurs de suppression (V3) - Index secondaires (filtres Bloom)

Voir aussi : Deletion Vector, Statistics

REST Catalog

Spécification d'API OpenAPI définissant une interface REST standardisée pour interagir avec les catalogues Iceberg. Permet l'interopérabilité entre clients et serveurs indépendamment de leur implémentation.

Implémentations : Apache Polaris, Project Nessie, AWS Glue, Google BigLake

Voir aussi : Catalog, OAuth 2.0

Row Lineage (Lignage des lignes)

Métadonnées au niveau des lignes introduites en V3 pour le traitement incrémental. Chaque ligne reçoit un identifiant unique (`_row_id`) et un numéro de séquence de dernière mise à jour (`_last_updated_sequence_number`).

Applications : CDC, vues matérialisées, traitement différentiel

Voir aussi : Sequence Number, Snapshot

Schema Evolution (Évolution de schéma)

Capacité de modifier le schéma d'une table (ajout, suppression, renommage de colonnes) sans réécrire les données existantes. Les identifiants de champs stables assurent la compatibilité entre versions de schéma.

Opérations supportées : - Ajout de colonnes (avec valeur par défaut en V3) - Suppression de colonnes - Renommage de colonnes - Réordonnancement de colonnes - Élargissement de types (int ⟶ bigint)

Voir aussi : Partition Evolution, Field ID

Sequence Number (Numéro de séquence)

Entier croissant assigné à chaque opération de commit, permettant d'ordonner les opérations et de déterminer l'applicabilité des fichiers de suppression. Un fichier de suppression s'applique uniquement aux fichiers de données avec un numéro de séquence inférieur ou égal.

Voir aussi : Snapshot, Delete File

Snapshot

État complet et cohérent d'une table Iceberg à un instant donné. Chaque snapshot contient un identifiant unique, une référence au snapshot parent, un horodatage et un pointeur vers sa liste de manifestes.

Voir aussi : Time Travel, Snapshot Expiration

Snapshot Expiration (Expiration des snapshots)

Processus de maintenance supprimant les snapshots anciens et les fichiers de données orphelins pour libérer l'espace de stockage. Configuré selon une politique de rétention (temps ou nombre de snapshots).

Voir aussi : Snapshot, Orphan Files

Snapshot Isolation (Isolation des snapshots)

Niveau d'isolation transactionnelle où les lectures voient un état cohérent de la table correspondant à un snapshot spécifique, indépendamment des écritures concurrentes.

Voir aussi : ACID, Optimistic Concurrency

Statistics (Statistiques)

Métadonnées collectées sur les données pour l'optimisation des requêtes. Iceberg maintient des statistiques au niveau des colonnes (min, max, null count, NaN count) dans les manifestes et des statistiques avancées (NDV, histogrammes) dans les fichiers Puffin.

Voir aussi : Puffin, Partition Pruning

Time Travel (Voyage dans le temps)

Fonctionnalité permettant d'interroger une table dans un état historique, soit par identifiant de snapshot, soit par horodatage. Essentiel pour l'audit, le débogage et la reproductibilité analytique.

Syntaxe SQL :

```
SELECT * FROM table AS OF TIMESTAMP '2025-01-01';
SELECT * FROM table AS OF VERSION 12345678901234;
```

Voir aussi : Snapshot, Rollback

Variant

Type de données semi-structuré introduit en V3 pour stocker des données JSON-like avec un ensemble élargi de types primitifs (date, timestamp, binary, decimal). Permet l'ingestion de données non typées sans enforcement strict de schéma.

Voir aussi : Schema Evolution, V3

B.2 Terminologie Lakehouse et Data Engineering

ACID

Acronyme pour Atomicité, Cohérence (Consistency), Isolation et Durabilité. Propriétés transactionnelles garantissant l'intégrité des données. Iceberg implémente des transactions ACID via le contrôle optimiste de concurrence et l'immutabilité des fichiers.

Voir aussi : Snapshot Isolation, Optimistic Concurrency

CDC (Change Data Capture)

Processus capturant les modifications de données (insertions, mises à jour, suppressions) depuis une source pour les propager vers des systèmes cibles. Iceberg supporte le CDC via les fichiers de suppression et le lignage des lignes.

Voir aussi : Row Lineage, Merge-on-Read

Column Pruning (Élagage de colonnes)

Optimisation lisant uniquement les colonnes requises par une requête, exploitant les formats de fichiers colonnaires comme Parquet. Réduit significativement les I/O et la mémoire utilisée.

Voir aussi : File Pruning, Parquet

Data Lake

Architecture de stockage centralisée acceptant des données brutes de tous formats (structurées, semi-structurées, non structurées) à grande échelle. Typiquement basé sur du stockage objet comme Amazon S3, Azure ADLS ou Google Cloud Storage.

Voir aussi : Data Lakehouse, Object Storage

Data Lakehouse

Architecture hybride combinant la flexibilité et le coût du Data Lake avec les capacités transactionnelles et la performance du Data Warehouse. Apache Iceberg est un format de table fondamental pour les architectures Lakehouse.

Caractéristiques : Transactions ACID, évolution de schéma, performance optimisée, stockage ouvert

Voir aussi : Data Lake, Apache Iceberg

Data Warehouse

Système de stockage et d'analyse de données structurées optimisé pour les requêtes analytiques. Traditionnellement basé sur des architectures propriétaires avec stockage et calcul couplés.

Voir aussi : Data Lakehouse, OLAP

ELT (Extract, Load, Transform)

Pattern d'intégration de données chargeant les données brutes dans la destination avant transformation, exploitant la puissance de calcul du système cible. Prédominant dans les architectures Lakehouse modernes.

Voir aussi : ETL, Data Pipeline

ETL (Extract, Transform, Load)

Pattern d'intégration de données transformant les données dans un système intermédiaire avant chargement dans la destination. Pattern traditionnel des entrepôts de données.

Voir aussi : ELT, Data Pipeline

File Format (Format de fichier)

Spécification définissant comment les données sont sérialisées et stockées dans des fichiers individuels. Iceberg supporte Parquet, ORC et Avro comme formats de fichiers de données.

Voir aussi : Table Format, Parquet, ORC, Avro

File Pruning (Élagage de fichiers)

Optimisation excluant les fichiers de données non pertinents de la lecture basée sur les statistiques stockées dans les manifestes (min/max, null count). Réduit les I/O en évitant la lecture de fichiers entiers.

Voir aussi : Partition Pruning, Column Pruning

Infonuagique (Cloud Computing)

Modèle de fourniture de services informatiques (calcul, stockage, réseau) à la demande via internet. Les architectures Lakehouse modernes s'appuient sur l'infrastructure infonuagique pour la scalabilité et la flexibilité.

Voir aussi : Object Storage, Serverless

Medallion Architecture (Architecture Médaillon)

Pattern d'organisation des données en couches progressives de raffinement : Bronze (données brutes), Silver (données nettoyées et conformées), Gold (données agrégées et optimisées pour la consommation).

Voir aussi : Data Lakehouse, ELT

Object Storage (Stockage objet)

Service de stockage infonuagique gérant les données comme objets avec métadonnées associées. Offre durabilité, scalabilité et coût optimisé pour le stockage de données à grande échelle.

Exemples : Amazon S3, Azure Blob Storage, Google Cloud Storage, MinIO

Voir aussi : Infonuagique, Data Lake

OLAP (Online Analytical Processing)

Catégorie de traitement de données optimisée pour les requêtes analytiques complexes sur de grands volumes de données. Les tables Iceberg sont conçues pour les charges de travail OLAP.

Voir aussi : Data Warehouse, Analytics

Optimistic Concurrency (Concurrence optimiste)

Modèle de contrôle de concurrence où les transactions procèdent sans verrouillage, avec détection des conflits au moment du commit. Iceberg utilise la concurrence optimiste pour les écritures parallèles.

Voir aussi : ACID, Snapshot Isolation

Parquet

Format de fichier colonnaire open source optimisé pour les charges de travail analytiques. Format de données par défaut recommandé pour Iceberg, offrant compression efficace et lecture sélective de colonnes.

Voir aussi : ORC, Avro, File Format

Query Engine (Moteur de requête)

Système exécutant des requêtes SQL ou similaires sur les données. Plusieurs moteurs peuvent interroger les mêmes tables Iceberg : Apache Spark, Trino, Dremio, Flink, Presto.

Voir aussi : SQL, Apache Spark

Schema-on-Read

Approche où le schéma est appliqué lors de la lecture plutôt que lors de l'écriture. Permet la flexibilité d'ingestion mais peut causer des problèmes de qualité de données.

Voir aussi : Schema-on-Write, Data Lake

Schema-on-Write

Approche où le schéma est appliqué lors de l'écriture, validant les données avant stockage. Iceberg supporte l'enforcement de schéma à l'écriture tout en permettant l'évolution de schéma.

Voir aussi : Schema-on-Read, Schema Evolution

Table Format (Format de table)

Spécification définissant comment les données sont organisées, gérées et interrogées à un niveau d'abstraction supérieur aux fichiers individuels. Définit les métadonnées, les transactions et l'évolution de schéma.

Exemples : Apache Iceberg, Delta Lake, Apache Hudi

Voir aussi : File Format, Apache Iceberg

Write Amplification (Amplification d'écriture)

Phénomène où une modification logique mineure entraîne une réécriture physique disproportionnée. Le mode Copy-on-Write souffre d'amplification d'écriture; Merge-on-Read la minimise.

Voir aussi : Copy-on-Write, Merge-on-Read

B.3 Terminologie Streaming et Kafka

Apache Kafka

Plateforme de streaming d'événements distribuée pour la construction de pipelines de données temps réel et d'applications de streaming. Kafka peut alimenter des tables Iceberg via des connecteurs dédiés.

Voir aussi : Kafka Connect, Streaming Lakehouse

Batch Processing (Traitement par lots)

Mode de traitement de données collectées et traitées en groupes à intervalles définis. Complémentaire au traitement en flux (streaming) dans les architectures modernes.

Voir aussi : Stream Processing, Lambda Architecture

Broker

Serveur Kafka individuel gérant le stockage et la distribution des messages au sein d'un cluster. Les brokers forment l'épine dorsale de l'infrastructure Kafka.

Voir aussi : Apache Kafka, Partition (Kafka)

Consumer

Application ou service lisant des messages depuis des topics Kafka. Les consumers Iceberg écrivent les données consommées dans des tables Iceberg.

Voir aussi : Producer, Consumer Group

Consumer Group (Groupe de consommateurs)

Ensemble de consumers partageant la charge de lecture d'un topic Kafka. Chaque partition est assignée à un seul consumer du groupe, permettant le parallélisme.

Voir aussi : Consumer, Partition (Kafka)

Event Sourcing

Pattern architectural stockant l'état de l'application comme une séquence d'événements immuables. Les événements peuvent être matérialisés dans des tables Iceberg pour l'analyse.

Voir aussi : CDC, Streaming Lakehouse

Exactly-Once Semantics (Sémantiques exactly-once)

Garantie de traitement où chaque message est traité exactement une fois, sans perte ni duplication. Iceberg supporte les sémantiques exactly-once via les transactions ACID.

Voir aussi : At-Least-Once, ACID

Kafka Connect

Framework pour connecter Kafka à des systèmes externes via des connecteurs standardisés. Le connecteur Iceberg sink écrit les données Kafka dans des tables Iceberg.

Voir aussi : Apache Kafka, Sink Connector

Lambda Architecture

Architecture hybride combinant traitement par lots et traitement en flux pour gérer de grands volumes de données. Les tables Iceberg peuvent servir de couche de service unifiée.

Voir aussi : Kappa Architecture, Streaming Lakehouse

Kappa Architecture

Architecture simplifiée utilisant uniquement le traitement en flux pour tous les cas d'usage, éliminant la complexité du traitement par lots séparé.

Voir aussi : Lambda Architecture, Stream Processing

Offset

Position d'un message dans une partition Kafka, utilisée pour le suivi de la progression de lecture. Les offsets permettent la reprise après échec et le retraitement.

Voir aussi : Partition (Kafka), Consumer

Partition (Kafka)

Unité de parallélisme et d'ordonnancement dans un topic Kafka. Chaque partition est une séquence ordonnée de messages stockée sur un broker.

Voir aussi : Topic, Broker

Producer

Application ou service publiant des messages dans des topics Kafka. Les producers sont les sources de données dans l'architecture de streaming.

Voir aussi : Consumer, Topic

Sink Connector

Connecteur Kafka Connect écrivant des données depuis Kafka vers un système externe. Le sink connector Iceberg écrit dans des tables Iceberg avec gestion des transactions.

Voir aussi : Kafka Connect, Source Connector

Source Connector

Connecteur Kafka Connect lisant des données depuis un système externe vers Kafka. Utilisé pour la capture de données depuis bases de données, fichiers ou APIs.

Voir aussi : Kafka Connect, Sink Connector

Stream Processing (Traitement en flux)

Mode de traitement des données en continu à mesure de leur arrivée, permettant des réponses temps réel ou quasi-temps réel. Apache Flink est un moteur de stream processing populaire avec Iceberg.

Voir aussi : Batch Processing, Apache Flink

Streaming Lakehouse

Architecture combinant le streaming temps réel (Kafka) avec le stockage analytique (Iceberg) pour des analyses unifiées sur données fraîches et historiques.

Voir aussi : Data Lakehouse, Apache Kafka, Apache Iceberg

Topic

Canal de publication-souscription dans Kafka où les messages sont organisés. Chaque topic est divisé en partitions pour le parallélisme et la scalabilité.

Voir aussi : Partition (Kafka), Producer, Consumer

Watermark

Mécanisme de gestion du temps événementiel dans le traitement de flux, indiquant la progression du temps dans le flux de données. Utilisé pour gérer les données tardives et déclencher les calculs.

Voir aussi : Stream Processing, Event Time

B.4 Acronymes et abréviations

Acronyme	Signification	Description
ACID	Atomicity, Consistency, Isolation, Durability	Propriétés transactionnelles garantissant l'intégrité des données
ADLS	Azure Data Lake Storage	Service de stockage objet Microsoft Azure
API	Application Programming Interface	Interface de programmation applicative
AWS	Amazon Web Services	Services infonuagiques Amazon
BI	Business Intelligence	Intelligence d'affaires, analyse décisionnelle
CDC	Change Data Capture	Capture des modifications de données
CDO	Chief Data Officer	Directeur des données
CoW	Copy-on-Write	Mode d'écriture avec copie
CPU	Central Processing Unit	Unité centrale de traitement
CSV	Comma-Separated Values	Valeurs séparées par des virgules
DDL	Data Definition Language	Langage de définition de données
DML	Data Manipulation Language	Langage de manipulation de données
DV	Deletion Vector	Vecteur de suppression
ELT	Extract, Load, Transform	Pattern d'intégration données
EMR	Elastic MapReduce	Service AWS de traitement de données
ETL	Extract, Transform, Load	Pattern d'intégration données
GCP	Google Cloud Platform	Services infonuagiques Google
GCS	Google Cloud Storage	Service de stockage objet Google
HDFS	Hadoop Distributed File System	Système de fichiers distribué Hadoop
HMS	Hive Metastore	Métastore Apache Hive
HTTP	Hypertext Transfer Protocol	Protocole de transfert hypertexte
I/O	Input/Output	Entrées/Sorties
JDBC	Java Database Connectivity	Connectivité base de données Java
JSON	JavaScript Object Notation	Notation d'objets JavaScript
MB	Megabyte	Mégaoctet
ML	Machine Learning	Apprentissage automatique
MoR	Merge-on-Read	Mode d'écriture avec fusion à la lecture
NDV	Number of Distinct Values	Nombre de valeurs distinctes
[illegible]	Règle[illegible] Données	[illegible] protection des données

Index des termes par thème

Architecture et stockage

- Data Lake, Data Lakehouse, Data Warehouse
- Object Storage, HDFS, Infonuagique
- Medallion Architecture, Table Format, File Format

Apache Iceberg - Core

- Catalog, Metadata File, Manifest File, Manifest List
- Snapshot, Time Travel, Rollback
- Schema Evolution, Partition Evolution, Hidden Partitioning

Apache Iceberg - Opérations

- Copy-on-Write, Merge-on-Read
- Delete File, Deletion Vector, Position Delete, Equality Delete
- Compaction, Snapshot Expiration

Apache Iceberg - Avancé

- REST Catalog, Puffin, Row Lineage
- Statistics, Sequence Number, Variant

Performance et optimisation

- Partition Pruning, File Pruning, Column Pruning
- Write Amplification, ACID, Optimistic Concurrency

Streaming et Kafka

- Apache Kafka, Topic, Partition (Kafka)
- Producer, Consumer, Consumer Group
- Kafka Connect, Sink Connector, Source Connector
- Stream Processing, Streaming Lakehouse

Intégration de données

- CDC, ETL, ELT
- Batch Processing, Stream Processing
- Event Sourcing, Exactly-Once Semantics

Références croisées vers les chapitres

Terme	Chapitres de référence
Apache Iceberg	Chapitres 1, 2, 3
Catalog	Chapitres 5, 6
Compaction	Chapitre 10
Copy-on-Write / Merge-on-Read	Chapitres 3, 8
Deletion Vector	Chapitres 3, 10
Hidden Partitioning	Chapitres 2, 4
Manifest / Metadata	Chapitre 2
REST Catalog	Chapitres 5, 6
Schema Evolution	Chapitres 2, 4
Snapshot / Time Travel	Chapitres 2, 3
Statistics / Puffin	Chapitres 3, 10
Streaming Lakehouse	Chapitre 12

Résumé

Ce glossaire fournit les définitions normalisées de la terminologie utilisée dans le Volume IV et l'écosystème Lakehouse plus largement. Les termes sont organisés en quatre catégories principales :

Terminologie Apache Iceberg. Les concepts fondamentaux du format de table Iceberg, incluant la structure des métadonnées (catalog, snapshot, manifest), les mécanismes de mise à jour (Copy-on-Write, Merge-on-Read, delete files), et les fonctionnalités avancées (schema evolution, hidden partitioning, time travel).

Terminologie Lakehouse et Data Engineering. Les concepts architecturaux et les patterns de l'ingénierie des données moderne, des formats de fichiers aux architectures de référence en passant par les optimisations de performance.

Terminologie Streaming et Kafka. Les concepts du traitement de données en flux et de l'écosystème Apache Kafka, essentiels pour comprendre le Streaming Lakehouse présenté au Chapitre 12.

Acronymes et abréviations. Référence rapide des acronymes techniques utilisés dans l'ouvrage.

Ce glossaire doit être consulté comme référence complémentaire lors de la lecture des chapitres techniques, et comme outil de normalisation pour les équipes adoptant Apache Iceberg.

Dernière mise à jour : Janvier 2026

Volume IV — Apache Iceberg : Le Lakehouse Moderne

Collection « L'Entreprise Agentique »

André-Guy Bruneau · 2026

Document généré avec Typst