Recommandations et Stratégies

1. Optimisations Supplémentaires à Court Terme

1.1 Migration DECIMAL vers REAL

Problème : Le type DECIMAL est précis mais lent pour les calculs. Pour des données météorologiques, la précision à virgule flottante est suffisante.

Recommandation : Migrer les colonnes temperature et humidity vers REAL ou DOUBLE PRECISION. Ces types utilisent l'arithmétique native du CPU.

Gain estimé : 10-20% plus rapide sur les agrégations

Priorité : Moyenne

1.2 Ajout de Métriques de Performance

Problème : Pas de visibilité sur les performances en production. Difficile de détecter les régressions.

Recommandation : Implémenter un système de monitoring avec :

- Temps de réponse par endpoint (P50, P95, P99)
- Taux de hit du cache
- Utilisation du pool de connexions
- Métriques de base de données (slow queries)
- Alertes automatiques sur seuils

Solutions suggérées : Prometheus + Grafana, ou APM comme New Relic/DataDog

Priorité : Haute

1.3 Validation des Paramètres d'Entrée

Problème : Pas de validation des plages de dates. Un utilisateur peut demander 100 ans de données même avec pagination.

Recommandation Ajouter des limites :

- Plage de dates maximum : 1 an
- Validation que from < to
- Limite de offset pour éviter deep pagination (ex: offset < 10000)

Priorité : Moyenne

2. Architecture et Scalabilité à Moyen Terme

2.1 Cache Distribué (Redis)

Contexte : Le cache en mémoire actuel fonctionne bien pour une seule instance, mais ne scale pas horizontalement.

Recommandation : Migrer vers Redis pour :

- Cache partagé entre instances
- Persistance du cache (survie aux redémarrages)

- TTL automatique géré par Redis
- Possibilité d'utiliser Redis Cluster pour très haute disponibilité

Implémentation : Remplacer SimpleCache par un client Redis (ioredis), conserver la même interface

Priorité : Haute si scaling horizontal prévu

2.2 Partitionnement de Table (Table Partitioning)

Contexte : Avec des années de données, la table weather va devenir énorme (100M+ lignes). Même avec index, les performances vont dégrader.

Recommandation : Implémenter le partitionnement par date (range partitioning) :

- Partition par mois ou par trimestre
- PostgreSQL scanne seulement les partitions pertinentes
- Maintenance simplifiée (supprimer vieilles partitions)

Exemple :

```
CREATE TABLE weather (location VARCHAR, date TIMESTAMP, ...) PARTITION BY RANGE (date);  CREATE TABLE weather_2024_01 PARTITION OF weather FOR VALUES FROM ('2024-01-01') TO ('2024-02-01');
```

Priorité : Haute quand > 50M lignes

2.3 Read Replicas

Contexte : Les lectures (GET) sont beaucoup plus fréquentes que les écritures (POST). Une seule base de données devient un goulot.

Recommandation : Configurer des read replicas PostgreSQL :

- Master pour écritures (POST /data, POST /data/batch)
- Replicas pour lectures (GET /data, /avg, /max, /min)
- Load balancer pour distribuer lectures sur replicas

Gain estimé : Capacité de lecture × nombre de replicas

Priorité : Haute si > 1000 req/s

2.4 API Rate Limiting

Problème : Pas de protection contre l'abus. Un utilisateur malveillant peut saturer l'API.

Recommandation : Implémenter rate limiting avec express-rate-limit :

- Par IP : 100 requêtes/minute
- Par API key (si authentification) : 1000 requêtes/minute
- Endpoints coûteux (/data/batch) : limites plus strictes

Priorité : Haute si API publique

3. Best Practices pour le Long Terme

3.1 Politique de Rétention des Données

Recommandation : Définir une stratégie de rétention :

- Données détaillées : 2 ans
- Données agrégées (moyennes mensuelles) : 10 ans
- Archivage des vieilles données dans storage froid (S3)

Bénéfices : Réduction des coûts de stockage, meilleures performances

3.2 Tests de Performance Automatisés

Recommandation : Intégrer des tests de performance dans le CI/CD :

- Tests de charge avec Artillery ou k6
- Seuils de performance : échec si > 500ms
- Tests avant chaque déploiement

Bénéfices : Détection précoce des régressions

3.3 Documentation des Patterns

Recommandation : Documenter les patterns d'optimisation :

- Toujours faire les filtres en SQL
- Toujours utiliser les agrégations SQL
- Toujours paginer les résultats
- Toujours cacher les requêtes coûteuses

Bénéfices : Cohérence dans le code, onboarding plus rapide

3.4 Revue de Code Axée Performance

Recommandation : Checklist de revue de code :

- Les requêtes SQL utilisent-elles les index ?
- Y a-t-il du filtrage en JavaScript qui devrait être en SQL ?
- Les résultats sont-ils paginés ?
- Les requêtes coûteuses sont-elles cachées ?
- Les insertions par lot sont-elles utilisées ?

4. Problèmes Non Résolus

4.1 Streaming pour Très Grandes Requêtes

Problème : Même avec pagination, certains utilisateurs veulent télécharger des années de données. Charger 1000 lignes × 100 pages = problème mémoire.

Solution potentielle : Implémenter le streaming :

- Utiliser curseurs PostgreSQL
- Streamer les résultats en JSON line-delimited
- Client reçoit les données progressivement

Complexité : Élevée - Nécessite refonte du client et du serveur

4.2 Recherche Full-Text sur Locations

Limitation actuelle : Les requêtes doivent spécifier le nom exact de la location. Pas de recherche floue ("Lion" au lieu de "Lyon").

Solution potentielle : Implémenter PostgreSQL full-text search ou intégrer Elasticsearch pour recherche avancée.

Complexité : Moyenne

4.3 Agrégations Temporelles Complexes

Limitation actuelle : Pas de support pour moyennes mensuelles, tendances, prévisions.

Solution potentielle : Ajouter des endpoints :

- GET /weather/monthly-avg/:location
- GET /weather/trend/:location
- Utiliser window functions PostgreSQL

Complexité : Moyenne

5. Tableau de Priorités

Recommandation	Priorité	Complexité	Impact
Métriques performance	Haute	Faible	Élevé
Rate limiting	Haute	Faible	Élevé
Cache distribué (Redis)	Haute	Moyenne	Élevé
Partitionnement table	Haute	Moyenne	Élevé
Read replicas	Haute	Moyenne	Très élevé
Validation paramètres	Moyenne	Faible	Moyen
Migration DECIMAL → REAL	Moyenne	Faible	Moyen
Streaming	Faible	Élevée	Moyen

6. Conclusion

Les optimisations déjà implémentées ont résolu les problèmes critiques. Les recommandations ci-dessus visent à préparer l'application pour une croissance future et à maintenir des performances optimales sur le long terme.

Les priorités immédiates sont :

- Monitoring et métriques (visibilité)
- Rate limiting (protection)
- Redis pour scaling horizontal
- Partitionnement quand > 50M lignes