# RULES ENGINE — SPECIFICATION CANONIQUE

**Version 1.7.1**

> **Statut** : Canonique
> **Date** : 2026-01-07
> **Base** : SPEC v1.7.0 + Référence Consolidée v1.5.4→v1.6.0
> Ce document est la **référence normative unique** du moteur de règles.
> Toute implémentation DOIT se conformer à cette spécification.
> Les tests existants sont **informatifs** sauf mention contraire.

---

## TABLE DES MATIÈRES


1. Objectifs et Invariants
2. Modèle de Données
3. Règles
4. Tokens
5. Agrégateurs
6. Évaluation et Exécution
7. Gestion des Erreurs
8. Compilation et Normalisation
9. Runner JSON
10. Modes d'Exécution
11. Tests Normatifs
12. Annexes


---

## 1. OBJECTIFS ET INVARIANTS

### 1.1 Objectifs

Le moteur de règles a pour objectif de :

| Objectif | Description |
|---|---|
| **Évaluer** | des règles scalaires |
| **À partir de** | variables initialisées |
| **Dans un** | thread isolé |
| **De manière** | déterministe, paresseuse et performante |

### 1.2 Invariants Fondamentaux (NON NÉGOCIABLES)

Ces invariants sont **immuables** et constituent le socle du moteur :

| # | Invariant | Description |
|---|---|---|
| I1 | **Orchestration** | Le moteur orchestre l'évaluation |
| I2 | **Délégation** | Le moteur ne calcule JAMAIS |
| I3 | **SQL Server** | SQL Server effectue 100% des calculs |
| I4 | **Exécution directe** | Toute expression finale est exécutable telle quelle par SQL Server |
| I5 | **Neutralité** | Aucune interprétation sémantique par le moteur |

### 1.3 Principe Cardinal

> **« Le moteur orchestre ; SQL Server calcule. »**

Ce principe implique :

- Le moteur **n'interprète jamais** les expressions SQL
- Le moteur **ne calcule jamais** directement
- Pas de `IIF`, `COALESCE`, ou logique dans `{...}`
- Le moteur se limite à : résoudre les tokens, substituer les valeurs, déléguer l'exécution

---

# 2. MODÈLE DE DONNÉES

## 2.1 Modèle Atomique

**Invariant depuis v1.5.5** : Une clé = Une valeur (jamais de multi-lignes)

| Concept | Description |
|---|---|
| **Clé** | Identifiant unique (case-insensitive) |
| **Valeur** | Scalaire unique (`NVARCHAR(MAX)`) |
| **Multiplicité** | Obtenue par sélection LIKE sur plusieurs clés |

## 2.2 Table d'État (#ThreadState)

Structure normative de la table d'état du thread :

| Colonne | Type | Description |
|---|---|---|
| `SeqId` | INT IDENTITY | Ordre d'insertion (canonique) |
| `Key` | NVARCHAR(200) | Identifiant unique |
| `IsRule` | BIT | 0=variable, 1=règle |
| `State` | TINYINT | État d'évaluation |
| `ScalarValue` | NVARCHAR(MAX) | Valeur résolue |
| `ValueIsNumeric` | BIT | Indicateur de type |
| `ErrorCategory` | VARCHAR(50) | Catégorie d'erreur |
| `ErrorCode` | VARCHAR(50) | Code d'erreur |

## 2.3 Collation

**Obligatoire** : `SQL_Latin1_General_CP1_CI_AS` (Case-Insensitive)

Toutes les comparaisons de clés DOIVENT utiliser cette collation.

---

# 3. RÈGLES

### 3.1 Définition

Une règle est :

- Une **expression SQL** valide
- Contenant potentiellement des **tokens** `{}`
- Stockée dans `RuleDefinitions`

#### Contenu autorisé

| Élément | Exemple | Autorisé |
|---|---|---|
| Littéraux | `42`, `'texte'` | ✅ |
| Opérateurs SQL | `+`, `-`, `*`, `/`, `%` | ✅ |
| Fonctions SQL | `ROUND()`, `COALESCE()` | ✅ |
| Tokens | `{MONTANT}`, `{SUM(A%)}` | ✅ |
| Logique dans token | `{IIF(A>0,A,0)}` | ❌ |

### 3.2 Portée (Scope)

**Important** : Une règle **n'a aucun scope**.

Le scope n'existe **qu'au niveau du token**.

### 3.3 États d'une Règle

| État | Code | Description |
|---|---|---|
| `NOT_EVALUATED` | 0 | Jamais évaluée |
| `EVALUATING` | 1 | En cours d'évaluation |
| `EVALUATED` | 2 | Résultat disponible |
| `ERROR` | 3 | Erreur bloquante |

**Règles de transition**

```
NOT_EVALUATED → EVALUATING → EVALUATED
                           ↘ ERROR
```

- `NULL` n'est **pas** une erreur (State = EVALUATED avec ScalarValue = NULL)
- `ERROR` est **bloquant** et **propagé** aux dépendants
- Ré-entrée dans `EVALUATING` → `ERROR` (cycle détecté)

---

# 4. TOKENS

## 4.1 Définition

Le token est :

- La **seule unité interprétée** par le moteur
- Responsable de la **sélection**, de l'**agrégation** et de la **réduction scalaire**
- Encadré par `{` et `}`

**Opérations du Token**

Un token effectue **exactement quatre opérations** :

1. **Sélectionner** un sous-ensemble de clés (variables et/ou règles)
2. **Résoudre** les valeurs correspondantes (lazy pour les règles)
3. **Appliquer** un agrégateur unique
4. **Retourner** un scalaire

**Ce qu'un Token NE FAIT PAS**

- ❌ Aucune logique SQL évaluée dans `{...}`
- ❌ Pas de `IIF`, `COALESCE`, ou calcul
- ❌ Pas de manipulation de collections
- ❌ Pas de transformation sémantique

### 4.2 Grammaire Canonique

```bnf
Token       ::= "{" [ Whitespace ] Lookup [ Whitespace ] "}"

Lookup      ::= [ Aggregator "(" [ Whitespace ] ] Selector [ [ Whitespace ] ")" ]

Selector    ::= [ Scope ":" ] IdentifierOrPattern

Scope       ::= "var" | "rule" | "all"

IdentifierOrPattern ::= Identifier | Pattern

Pattern     ::= SqlLikeExpression

Aggregator  ::= "FIRST" | "LAST"
              | "SUM" | "AVG" | "MIN" | "MAX" | "COUNT"
              | "SUM_POS" | "SUM_NEG"
              | "COUNT_POS" | "COUNT_NEG"
              | "FIRST_POS" | "FIRST_NEG"
              | "LAST_POS" | "LAST_NEG"
              | "CONCAT" | "JSONIFY"

Whitespace  ::= ( " " | "\t" )*
```

#### Agrégateurs non normatifs (héritage)

Les agrégateurs suivants sont présents dans certaines implémentations mais **non garantis** :

- `AVG_POS` / `AVG_NEG`
- `MIN_POS` / `MIN_NEG`
- `MAX_POS` / `MAX_NEG`

### 4.3 Scope

#### Scopes disponibles

| Scope | Description | Sélectionne dans |
|---|---|---|
| `var` | Variables uniquement | Variables |
| `rule` | Règles uniquement | Résultats de règles |

| Scope | Description | Sélectionne dans |
|---|---|---|
| `all` | Union | Variables ∪ Règles |

**Scope par défaut**

**Défaut** : `all`

```
{MONTANT}     ≡ {FIRST(all:MONTANT)}   -- si MONTANT non numérique
{MONTANT}     ≡ {SUM(all:MONTANT)}     -- si MONTANT numérique
{var:A%}      → scope explicite var
{rule:R_%}    → scope explicite rule
```

### 4.4 Agrégateur par Défaut (NOUVEAU v1.7.1)

**Règle contextuelle**

L'agrégateur par défaut dépend du **type des valeurs sélectionnées** :

| Contexte | Agrégateur par défaut | Justification |
|---|---|---|
| **Valeurs numériques** | `SUM` | Comportement naturel pour les nombres |
| **Valeurs non numériques** | `FIRST` | SUM n'a pas de sens sur du texte |
| **Ensemble mixte** | `FIRST` | Comportement sûr par défaut |
| **Ensemble vide** | N/A | Retourne NULL (ou 0 pour COUNT) |

**Détermination du type**

La détermination se fait sur le **premier élément non-NULL** du ValueSet :

- Si `TRY_CAST(value AS DECIMAL(38,18))` réussit → numérique → `SUM`
- Sinon → non numérique → `FIRST`

**Exemples**

```sql
```

```
-- Variables numériques
{MONTANT_%}      ≡ {SUM(all:MONTANT_%)}    -- 100+200+150 = 450

-- Variables textuelles
{LIBELLE_%}      ≡ {FIRST(all:LIBELLE_%)}  -- 'A'

-- Référence directe numérique
{MONTANT_1}      ≡ {SUM(all:MONTANT_1)}    -- 100 (SUM d'un seul = lui-même)

-- Référence directe textuelle
{CONFIG}         ≡ {FIRST(all:CONFIG)}     -- valeur JSON

-- Recommandation : toujours expliciter l'agrégateur
{SUM(MONTANT_%)}  -- Explicite et clair
{FIRST(CONFIG)}   -- Explicite et clair
```

**Implémentation**

```sql
-- Pseudo-code de détermination
IF @Aggregator IS NULL  -- Pas d'agrégateur explicite
BEGIN
   -- Vérifier le type du premier élément non-NULL
   DECLARE @FirstValue NVARCHAR(MAX);
   SELECT TOP 1 @FirstValue = ScalarValue
   FROM #ThreadState
   WHERE [Key] LIKE @Pattern AND ScalarValue IS NOT NULL
   ORDER BY SeqId;

   IF TRY_CAST(@FirstValue AS DECIMAL(38,18)) IS NOT NULL
      SET @Aggregator = 'SUM';    -- Numérique → SUM
   ELSE
      SET @Aggregator = 'FIRST';  -- Non numérique → FIRST
END
```

## 4.5 Wildcards

**Syntaxes supportées**

| Syntaxe | Caractère | Description |
|---|---|---|
| SQL LIKE (canonique) | `%` | 0 à N caractères |

| Syntaxe | Caractère | Description |
|---|---|---|
| SQL LIKE (canonique) | `_` | Exactement 1 caractère |
| Alias utilisateur | `*` | Converti en `%` |
| Alias utilisateur | `?` | Converti en `_` |

#### Normalisation

Avant exécution, les wildcards utilisateur sont normalisés :

```
{A*}   → {A%}
{A?B}  → {A_B}
{R_*}  → {R_%}
```

### 4.6 Tolérance aux Espaces

Les espaces sont tolérés autour des éléments structurants :

```sql
{SUM(A%)}          -- Forme canonique
{ SUM(A%) }        -- Toléré
{SUM( var:A% )}    -- Toléré
{ SUM ( var : A% ) }-- Toléré (mais déconseillé)
```

**Recommandation** : Utiliser la forme canonique sans espaces superflus.

### 4.7 Identifiants

#### Règles

| Règle | Description |
|---|---|
| Espaces | Autorisés dans les identifiants |
| Quotes | `'...'` ou `"..."` pour échappement |
| Échappement interne | `''` dans `'...'`, `""` dans `"..."` |
| Caractères interdits | `{ } [ ] ( ) :` hors quotes |

#### Exemples

```sql
{MONTANT HT}            -- Identifiant avec espace
{'Clé avec {accolades}'}  -- Identifiant quoté
{"Valeur ""échappée"""}   -- Double-quote échappée
```

---

# 5. AGRÉGATEURS

## 5.1 Liste des Agrégateurs Normatifs

### Agrégateurs numériques

| Agrégateur | Description | NULL | Ensemble vide |
|---|---|---|---|
| `SUM` | Somme | Ignoré | NULL |
| `AVG` | Moyenne | Ignoré | NULL |
| `MIN` | Minimum | Ignoré | NULL |
| `MAX` | Maximum | Ignoré | NULL |
| `COUNT` | Compte non-NULL | Ignoré | **0** |

### Agrégateurs positionnels

| Agrégateur | Description | NULL | Ensemble vide |
|---|---|---|---|
| `FIRST` | Premier par SeqId | Ignoré | NULL |
| `LAST` | Dernier par SeqId | Ignoré | NULL |

### Agrégateurs filtrés (positifs)

| Agrégateur | Description | Filtre |
|---|---|---|
| `SUM_POS` | Somme des positifs | `value > 0` |
| `COUNT_POS` | Compte des positifs | `value > 0` |
| `FIRST_POS` | Premier positif | `value > 0` |

| Agrégateur | Description | Filtre |
|---|---|---|
| `LAST_POS` | Dernier positif | `value > 0` |

**Agrégateurs filtrés (négatifs)**

| Agrégateur | Description | Filtre |
|---|---|---|
| `SUM_NEG` | Somme des négatifs | `value < 0` |
| `COUNT_NEG` | Compte des négatifs | `value < 0` |
| `FIRST_NEG` | Premier négatif | `value < 0` |
| `LAST_NEG` | Dernier négatif | `value < 0` |

**Agrégateurs textuels**

| Agrégateur | Description | NULL | Ensemble vide |
|---|---|---|---|
| `CONCAT` | Concaténation ordonnée | Ignoré | `''` (vide) |
| `JSONIFY` | Objet JSON | Ignoré | `'{}'` |

## 5.2 Règles Communes

### Gestion des NULL

**Règle universelle (v1.6.0+)** : Tous les agrégateurs ignorent les valeurs NULL.

```
sql

-- Variables: MONTANT_1=100, MONTANT_2=NULL, MONTANT_3=200
{SUM(MONTANT_%)}   → 300 (NULL ignoré)
{COUNT(MONTANT_%)} → 2   (NULL ignoré)
{FIRST(MONTANT_%)} → 100 (NULL ignoré, premier non-NULL)
```

### Ordre canonique

**Strictement basé sur SeqId** (ordre d'insertion).

Aucun tri SQL implicite n'est autorisé. L'ordre est déterministe et reproductible.

```
sql
```

```
-- Insertion: LIBELLE_C, LIBELLE_A, LIBELLE_B (dans cet ordre)
{FIRST(LIBELLE_%)}  → 'C' (premier inséré)
{LAST(LIBELLE_%)}   → 'B' (dernier inséré)
{CONCAT(LIBELLE_%)} → 'CAB' (ordre SeqId)
```

**Ensemble vide**

| Agrégateur | Résultat ensemble vide |
|---|---|
| SUM, AVG, MIN, MAX | NULL |
| COUNT, COUNT_POS, COUNT_NEG | **0** |
| FIRST, LAST, FIRST_, *LAST_* | NULL |
| CONCAT | `''` (chaîne vide) |
| JSONIFY | `'{}'` (objet vide) |

## 5.3 CONCAT

**Spécification**

- Concaténation **sans séparateur** (v1.6.0+)
- Ordre strictement par SeqId
- NULL ignorés
- Unicode supporté

```
sql

-- LIBELLE_1='A', LIBELLE_2='B', LIBELLE_3=NULL, LIBELLE_4='C'
{CONCAT(LIBELLE_%)} → 'ABC'  -- NULL ignoré, pas de séparateur
```

**Implémentation**

```
sql

STRING_AGG(ScalarValue, '') WITHIN GROUP (ORDER BY SeqId)
```

### 5.4 JSONIFY

**Spécification**

- Génère un objet JSON `{"key1":value1,"key2":value2,...}`
- Ordre strictement par SeqId
- NULL ignorés (clé absente)
- Types préservés : nombres, booléens, chaînes, objets imbriqués
- Unicode supporté

```sql
-- A=1, B='text', C=NULL, D=true
{JSONIFY(all:%)} → '{"A":1,"B":"text","D":true}'  -- C ignoré
```

**Règles de typage**

| Type source | Rendu JSON |
|---|---|
| Numérique | `123` (sans quotes) |
| Booléen (`true`/`false`) | `true`/`false` (sans quotes) |
| Chaîne | `"valeur"` (avec quotes) |
| JSON valide | Inséré tel quel |
| NULL | Clé ignorée |

---

## 6. ÉVALUATION ET EXÉCUTION

### 6.1 Évaluation Paresseuse (Lazy)

**Lazy Rule**

Une règle n'est évaluée **que si requise** :

- Explicitement demandée dans `rules[]`
- Référencée par une autre règle via token

#### Lazy Token

Seules les règles **réellement matchées** par un pattern sont évaluées :

```sql
-- Si seules R_A et R_B existent et matchent R_%
{SUM(rule:R_%)}  -- Évalue R_A et R_B uniquement
```

#### Lazy Discovery

Les règles non présentes dans `#ThreadState` sont découvertes à la demande via `sp_DiscoverRulesLike`.

### 6.2 Dépendances

#### Dépendances dynamiques

Les dépendances sont résolues dynamiquement lors de l'évaluation des tokens.

```sql
-- R_A = {MONTANT_1} + {rule:R_B}
-- Dépendances de R_A : MONTANT_1, R_B
```

#### Détection de cycles

La détection de cycles est **obligatoire**.

| Type de cycle | Exemple | Résultat |
|---|---|---|
| Self-cycle | `R = {rule:R}+1` | ERROR |
| Cycle mutuel | `A = {rule:B}`, `B = {rule:A}` | ERROR |
| Cycle indirect | `A → B → C → A` | ERROR |

#### Self-match dans patterns

**Distinction importante** (v1.7.1) :

| Situation | Comportement |
|---|---|
| Self-match dans pattern | Ignoré (traité comme NULL) |
| Self-reference directe | ERROR (cycle) |

```sql
-- R_SUM = {SUM(rule:R_%)} où R_SUM matche R_%
-- → R_SUM s'ignore elle-même, agrège les autres

-- SELF = {rule:SELF}+1
-- → ERROR : self-cycle direct
```

### 6.3 Cache

**Cache de compilation**

- Stocke les expressions compilées (SQL normalisé + tokens parsés)
- Clé : hash de l'expression source
- Invalidation : modification de la règle

**Cache d'exécution**

- Stocke les résultats d'évaluation (`#ThreadState`)
- Durée de vie : thread courant
- Désactivé en mode DEBUG

---

## 7. GESTION DES ERREURS

### 7.1 Contrat Unifié

Toute erreur lors de l'évaluation d'une règle :

1. Marque la règle **ERROR** (State = 3)
2. Associe `ErrorCategory` et `ErrorCode`
3. Valeur scalaire = **NULL**
4. Le thread **continue** (pas d'arrêt global)

### 7.2 Catégories d'Erreurs

| Catégorie | Codes typiques | Description |
|---|---|---|
| RECURSION | CYCLE, SELF_CYCLE, MAX_DEPTH | Dépendance circulaire |
| NUMERIC | DIVIDE_BY_ZERO, OVERFLOW | Erreur arithmétique |
| TYPE | TYPE_MISMATCH, INVALID_CAST | Incompatibilité de type |
| SYNTAX | INVALID_EXPRESSION | Expression malformée |
| SQL | SQL_ERROR, EVAL_ERROR | Erreur SQL Server |
| RULE | NOT_FOUND | Règle inexistante |
| UNKNOWN | UNEXPECTED | Erreur non classifiée |

### 7.3 Propagation

#### Dans les agrégats

Les valeurs ERROR/NULL sont **ignorées** par tous les agrégateurs (v1.6.0+).

#### Dans les dépendances

Si une dépendance est en ERROR :

- L'erreur est **propagée**
- La règle dépendante passe en ERROR
- Le cycle de propagation continue

```sql
-- R_A = {rule:R_B} + 1
-- Si R_B = ERROR → R_A = ERROR (propagation)
```

---

## 8. COMPILATION ET NORMALISATION

### 8.1 Normalisation des Littéraux

Le compilateur normalise **avant exécution SQL** :

| Transformation | Avant | Après |
|---|---|---|
| Quotes | `"texte"` | `'texte'` |
| Séparateur décimal FR | `2,5` | `2.5` |
| Échappement quotes | `l'exemple` | `l''exemple` |

## 8.2 Normalisation des Résultats

### Résultats numériques

Suppression des zéros inutiles et normalisation :

```sql
'10.500000000' → '10.5'
'42.000000000' → '42'
'-0.000000000' → '0'
```

### Résultats textuels

Conservés intégralement (`NVARCHAR(MAX)`).

## 8.3 Normalisation des Tokens

### Wildcards

```
{A*}    → {A%}
{A?B}   → {A_B}
```

### Espaces

Les espaces superflus sont supprimés lors du parsing (LTRIM/RTRIM).

## 8.4 Forme Canonique

Toute expression est convertie en forme canonique avant stockage en cache :

```sql
```

```
-- Forme utilisateur
{ SUM( var : MONTANT_* ) }

-- Forme canonique
{SUM(var:MONTANT_%)}
```

---

# 9. RUNNER JSON

## 9.1 Rôle

Le runner JSON est un **orchestrateur neutre**. Il :

1. Initialise le thread (`#ThreadState`, `#CallStack`, etc.)
2. Charge les variables atomiques
3. Exécute une **liste explicite** de règles
4. Retourne les résultats au format JSON

## 9.2 Ce que le Runner NE FAIT PAS

- ❌ N'interprète pas les tokens
- ❌ N'applique pas d'agrégateurs
- ❌ Ne résout pas de dépendances
- ❌ N'utilise pas `rule:` ni patterns dans `rules[]`

## 9.3 Schéma JSON d'Entrée

```json
```

```json
{
 "mode": "NORMAL",
 "variables": [
  { "key": "MONTANT_1", "type": "DECIMAL", "value": "100" },
  { "key": "MONTANT_2", "type": "DECIMAL", "value": "200" },
  { "key": "CONFIG", "type": "JSON", "value": "{\"threshold\":50}" }
 ],
 "rules": ["RULE_A", "RULE_B", "RULE_C"],
 "options": {
  "stopOnFatal": false,
  "returnStateTable": true,
  "returnDebug": false
 }
}
```

**Contraintes**

| Élément | Contrainte |
|---|---|
| `mode` | `"NORMAL"` ou `"DEBUG"` |
| `variables[].key` | Unique (case-insensitive) |
| `variables[].value` | Scalaire texte |
| `rules[]` | Liste de **codes de règles** (pas de patterns) |

### 9.4 Schéma JSON de Sortie

```json
```

```
{
 "success": true,
 "mode": "NORMAL",
 "summary": {
  "totalRules": 3,
  "evaluated": 3,
  "errors": 0
 },
 "results": [
  { "ruleCode": "RULE_A", "value": "150", "state": "EVALUATED" },
  { "ruleCode": "RULE_B", "value": null, "state": "ERROR", "errorCode": "DIVIDE_BY_ZERO" }
 ],
 "stateTable": [...],
 "debug": [...]
}
```


---

# 10. MODES D'EXÉCUTION

## 10.1 Mode NORMAL (Défaut)

**Objectif** : Performance maximale

| Aspect | Comportement |
|---|---|
| Journalisation | Minimale |
| Stockage | State, Value, ErrorCategory, ErrorCode |
| Cache compilation | Activé |
| Cache exécution | Activé |

## 10.2 Mode DEBUG

**Objectif** : Diagnostic complet

| Aspect | Comportement |
|---|---|
| Journalisation | Complète (#ThreadDebug) |
| Stockage | Tout + SQL compilé, durées |
| Cache compilation | Activé (avec hits/miss) |
| Cache exécution | **Désactivé** |

**Informations DEBUG**

- Durée d'évaluation par règle
- SQL compilé final
- Ordre d'évaluation
- Tokens résolus
- Erreurs détaillées

---

# 11. TESTS NORMATIFS

## 11.1 Fixtures Standard

**Variables MONTANT_%**

| SeqId | Key | ScalarValue | Type |
|---|---|---|---|
| 1 | MONTANT_1 | 100 | Numérique |
| 2 | MONTANT_2 | 200 | Numérique |
| 3 | MONTANT_3 | -50 | Numérique |
| 4 | MONTANT_4 | 150 | Numérique |
| 5 | MONTANT_5 | -25 | Numérique |
| 6 | MONTANT_6 | NULL | NULL |

**Variables LIBELLE_%**

| SeqId | Key | ScalarValue | Type |
|---|---|---|---|
| 1 | LIBELLE_1 | 'A' | Texte |
| 2 | LIBELLE_2 | 'B' | Texte |
| 3 | LIBELLE_3 | NULL | NULL |
| 4 | LIBELLE_4 | 'C' | Texte |

## 11.2 Matrice de Tests

**Agrégateurs par défaut (v1.7.1)**

| Test | Expression | Attendu | Justification |
|---|---|---|---|
| D01 | {MONTANT_1} | 100 | Numérique → SUM (= valeur) |
| D02 | {MONTANT_%} | 375 | Numérique → SUM |
| D03 | {LIBELLE_1} | 'A' | Texte → FIRST |
| D04 | {LIBELLE_%} | 'A' | Texte → FIRST |

**Agrégateurs explicites**

| Test | Expression | Attendu |
|---|---|---|
| A01 | {SUM(MONTANT_%)} | 375 |
| A02 | {SUM_POS(MONTANT_%)} | 450 |
| A03 | {SUM_NEG(MONTANT_%)} | -75 |
| A04 | {AVG(MONTANT_%)} | 75 |
| A05 | {COUNT(MONTANT_%)} | 5 |
| A06 | {MIN(MONTANT_%)} | -50 |
| A07 | {MAX(MONTANT_%)} | 200 |

### Ordre canonique

| Test | Expression | Attendu |
|---|---|---|
| O01 | {FIRST(MONTANT_%)} | 100 |
| O02 | {LAST(MONTANT_%)} | -25 |
| O03 | {FIRST_NEG(MONTANT_%)} | -50 |
| O04 | {LAST_POS(MONTANT_%)} | 150 |
| O05 | {CONCAT(LIBELLE_%)} | 'ABC' |

### Gestion NULL (v1.6.0+)

| Test | Expression | Attendu | Note |
|---|---|---|---|
| N01 | {SUM(MONTANT_%)} | 375 | NULL ignoré |
| N02 | {COUNT(MONTANT_%)} | 5 | NULL ignoré |
| N03 | {FIRST(LIBELLE_%)} | 'A' | NULL ignoré |

### Ensemble vide

| Test | Expression | Attendu |
|---|---|---|
| E01 | {SUM(INEXISTANT_%)} | NULL |
| E02 | {COUNT(INEXISTANT_%)} | 0 |
| E03 | {CONCAT(INEXISTANT_%)} | '' |
| E04 | {JSONIFY(INEXISTANT_%)} | '{}' |

# 12. ANNEXES

## 12.1 Contrat IA-First

### Interdictions

- ❌ Ne pas inventer de grammaire ou d'agrégateur
- ❌ Ne pas évaluer SQL dans `{...}`
- ❌ Ne pas changer l'ordre canonique (SeqId)
- ❌ Ne pas stopper le thread en cas d'erreur

### Obligations

- ✅ États fermés : NOT_EVALUATED, EVALUATING, EVALUATED, ERROR
- ✅ Ré-entrée EVALUATING → ERROR + NULL
- ✅ Stockage NVARCHAR(MAX)
- ✅ Collation case-insensitive

## 12.2 Historique des Versions

| Version | Date | Changements majeurs |
|---|---|---|
| v1.5.4 | 2025-12-18 | Fondation sémantique |
| v1.5.5 | 2025-12-19 | Modèle atomique explicite, Runner JSON |
| v1.6.0 | 2025-12-20 | NULL ignorés uniformément, LAST ajouté |
| v1.7.0 | 2026-01-06 | Grammaire canonique, SUM par défaut |
| **v1.7.1** | **2026-01-07** | **Agrégateur par défaut contextuel (SUM/FIRST)** |

## 12.3 Glossaire

| Terme | Définition |
|---|---|
| **Token** | Unité `{...}` de résolution → scalaire |
| **Thread** | Contexte isolé d'évaluation |
| **SeqId** | Ordre d'insertion (ordre canonique) |

| Terme | Définition |
|---|---|
| **Lazy** | Évaluation à la demande |
| **Scope** | Filtre de sélection (var/rule/all) |
| **ValueSet** | Ensemble transitoire avant agrégation |
| **Pattern** | Expression LIKE pour sélection multiple |

### 12.4 Références

| Document | Rôle |
|---|---|
| REFERENCE_v1.5.4.md | Fondation sémantique |
| REFERENCE_v1.5.5.md | Modèle atomique |
| REFERENCE_v1.6.0.md | Simplification NULL |
| RULES_ENGINE_REFERENCE_CONSOLIDEE.md | Traçabilité complète |
| SPEC_v1.7.0.md | Grammaire canonique |

## FIN DU DOCUMENT

**Version** : 1.7.1
**Statut** : Canonique
**Date** : 2026-01-07